UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MP-DRAUGHTS - UM SISTEMA MULTIAGENTE DE APRENDIZAGEM AUTOMÁTICA PARA DAMAS BASEADO EM REDES NEURAIS DE KOHONEN E *PERCEPTRON* MULTICAMADAS

VALQUÍRIA APARECIDA ROSA DUARTE

Uberlândia - Minas Gerais
2009

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VALQUÍRIA APARECIDA ROSA DUARTE

MP-DRAUGHTS - UM SISTEMA MULTIAGENTE DE APRENDIZAGEM AUTOMÁTICA PARA DAMAS BASEADO EM REDES NEURAIS DE KOHONEN E *PERCEPTRON* MULTICAMADAS

Dissertação de Mestrado apresentada à Faculdade de Ciência da Computação da Universidade Federal de Uberlândia, Minas Gerais, como parte dos requisitos exigidos para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Inteligência Artificial.

Orientadora:
Prof^a. Dr^a. Rita Maria da Silva Julia

Uberlândia, Minas Gerais
2009

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Os abaixo assinados, por meio deste, certificam que leram e recomendam para a Faculdade de Ciência da Computação a aceitação da dissertação intitulada "***MP-Draughts* - Um Sistema Multiagente de Aprendizagem Automática para Damas Baseado em Redes Neurais de Kohonen e *Perceptron* Multicamadas** " por **Valquíria Aparecida Rosa Duarte** como parte dos requisitos exigidos para a obtenção do título de **Mestre em Ciência da Computação**.

Uberlândia, 17 de Julho de 2009

Orientadora: _____

Prof^a. Dr^a. Rita Maria da Silva Julia
Universidade Federal de Uberlândia

Banca Examinadora:

_____

Prof. Dr. Ivan Nunes da Silva
Universidade de São Paulo - USP

_____

Prof. Dr. Keiji Yamanaka
Universidade Federal de Uberlândia - UFU

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Data: Julho de 2009

Autor:	**Valquíria Aparecida Rosa Duarte**
Título:	***MP-Draughts* - Um Sistema Multiagente de Aprendizagem Automática para Damas Baseado em Redes Neurais de Kohonen e *Perceptron* Multicamadas**
Faculdade:	**Faculdade de Ciência da Computação**
Grau:	**Mestrado**

Fica garantido à Universidade Federal de Uberlândia o direito de circulação e impressão de cópias deste documento para propósitos exclusivamente acadêmicos, desde que o autor seja devidamente informado.

_____

Autor

Dedicatória

À minha mãe Maria Aparecida Rosa, à minha irmã Vanessa do Rosário Rosa, às minhas tias, primas, primos e minhas grandes amizades. Pessoas que sempre me apoiaram e caminharam comigo na busca pelos meus ideais, sendo meu alicerce e meu ânimo para nunca desistir dos meus sonhos nem fraquejar diante dos obstáculos.

Agradecimentos

Agradeço...

Primeiramente a Deus, pelo dom da vida, por ser meu guia em todos os momentos da minha vida...

A minha mãe pela dedicação, confiança, apoio, amor incondicional, pela educação e pelos valores a mim ensinados.

A toda a minha família que sempre esteve comigo, confiando em mim, na minha capacidade de lutar e buscar meus ideais, mesmo que, às vezes, não concordassem plenamente com as minhas escolhas. Obrigada pela confiança!

Aos professores do Departamento de Pós-Graduação da FACOM/UFU pelo conhecimento compartilhado durante todo o período do mestrado. Também à ex-secretária da Pós-Graduação Maria Helena e ao atual secretário, Erisvaldo, pela gentileza, carinho e amizade com que sempre me atenderam.

Principalmente, à Professora Rita Maria da Silva Julia pela dedicação, eficiência, gentileza e profissionalismo com que conduziu sua orientação no desenvolver deste trabalho. Agradeço-lhe ainda pelo apoio, incentivo, paciência, amizade e ensinamentos de vida.

Aos meus amigos que deixei em Catalão e que mesmo de longe sempre me incentivaram e me apoiaram. Obrigada Jack, Tó, Lara, Jana, Claitin, Dani, Dayane...

A Eleni que, aqui em Uberlândia, foi a pessoa com quem pude contar e que sempre se dispôs a me ajudar.

Ao meu amigo do Laboratório de Inteligência Artificial, Robson Lopes da Silva, que se mostrou amigo, companheiro e que, direta/indiretamente, colaborou para a realização deste trabalho. Agradeço também aos outros amigos de mestrado de outras linhas de pesquisa, Nubia Rosa, Liliane do Nascimento, Tauller Augusto, Rodrigo Reis, Marcos Roberto, Walter Alexandre e todos os outros que estiveram comigo nesse período.

A CAPES, pelo apoio financeiro concedido para a realização deste trabalho.

De um modo geral obrigada a todos que contribuíram, de alguma forma, para a concretização deste trabalho e também para a minha formação como pessoa.

Muito Obrigada!

"Quero, um dia, dizer às pessoas que nada foi em vão... Que o amor existe, que vale a pena se doar às amizades e às pessoas, que a vida é bela sim e que eu sempre dei o melhor de mim... e que valeu a pena." (Mário Quintana)

Resumo

O objetivo deste trabalho é propor um sistema de aprendizagem de Damas, o *MP-Draughts* (*MultiPhase- Draughts*): um sistema multiagentes, em que um deles - conhecido como *IIGA (Initial/Intermediate Game Agent)*- é desenvolvido e treinado para ser especializado em fases iniciais e intermediárias de jogo e os outros 25 agentes, em fases finais. Cada um dos agentes que compõe o *MP-Draughts* é uma rede neural que aprende a jogar com o mínimo possível de intervenção humana (distintamente do agente campeão do mundo *Chinook*). O *MP-Draughts* é fruto de uma contínua atividade de pesquisa que teve como produto anterior o *VisionDraughts*. Apesar de sua eficiência geral, o *VisionDraughts*, muitas vezes, tem seu bom desempenho comprometido na fase de finalização de partidas, mesmo estando em vantagem no jogo em comparação com o seu oponente (por exemplo, entrando em *loop* de final de jogo). No sentido de reduzir o comportamento indesejado do jogador, o *MP-Draughts* conta com 25 agentes especializados em final de jogo, sendo que cada um é treinado para lidar com um determinado tipo de *cluster* de tabuleiros de final de jogo. Esses 25 clusters são minerados por redes de *Kohonen-SOM* de uma base de dados que contém uma grande quantidade de estado de tabuleiro de final de jogo. Depois de treinado, o *MP-Draughts* atua da seguinte maneira: primeiro, uma versão aprimorada do *VisionDraughts* é usada como o *IIGA*; depois, um agente de final de jogo que representa o cluster que mais se aproxima do estado corrente do tabuleiro do jogo deverá substituir o *IIGA* e conduzir o jogo até o final. Este trabalho mostra que essa estratégia melhorou, significativamente, o desempenho geral do agente jogador.

Palavras chave: sistemas multi-agentes; algoritmos de clusterização; redes neurais artificiais; aprendizagem por reforço; aprendizagem por diferenças temporais; busca eficiente; jogos;

Abstract

The goal of this work is to present MP-Draughts (MultiPhase- Draughts), that is a multiagent environment for Draughts, where one agent - named IIGA- is built and trained such as to be specialized for the initial and the intermediate phases of the games and the remaining ones for the final phases of them. Each agent of MP-Draughts is a neural network which learns almost without human supervision (distinctly from the world champion agent Chinook). MP-Draughts issues from a continuous activity of research whose previous product was the efficient agent VisionDraughts. Despite its good general performance, VisionDraughts frequently does not succeed in final phases of a game, even being in advantageous situation compared to its opponent (for instance, getting into endgame loops). In order to try to reduce this misbehavior of the agent during endgames, MP-Draughts counts on 25 agents specialized for endgame phases, each one trained such as to be able to deal with a determined cluster of endgame boardstates. These 25 clusters are mined by a Kohonen-SOM Network from a Data Base containing a large quantity of endgame boardstates. After trained, MP-Draughts operates in the following way: first, an optimized version of VisionDraughts is used as IIGA; next, the endgame agent that represents the cluster which better fits the current endgame board-state will replace it up to the end of the game. This work shows that such a strategy significantly improves the general performance of the player agents.

Keywords: multiagent system; clustering algorithm; artificial neural network; reinforcement learning; temporal difference learning; network; efficient search; game

Sumário

Lista de Figuras **xxi**

Lista de Tabelas **xxiii**

1 Introdução **25**

1.1 Introdução e Motivação 25

1.2 Estrutura da dissertação 27

2 Referencial Teórico **29**

2.1 Sistemas Multiagentes 29

2.1.1 Sistemas Multiagentes e o Conceito de Agente 30

2.2 *Clusterização* 31

2.2.1 Medida de Similaridade 33

2.2.2 Técnicas de *Clusterização* 33

2.3 Redes Neurais 35

2.3.1 A modelagem Matemática do Neurônio 36

2.3.2 Tipos de Redes Neurais 37

2.3.3 O *Perceptron* Simples e *Perceptron* Multicamadas - MLP 39

2.3.4 Redes Neurais Auto-Organizáveis - SOM 40

2.4 Aprendizagem por Reforço 43

2.4.1 Método de Solução por Diferenças Temporais 44

2.5 Estratégia de Busca Eficiente 45

2.5.1 Algoritmo de Busca Alfa-Beta 45

2.5.2 Algoritmo de Busca Alfa-Beta e Aprofundamento Iterativo 47

2.5.3 Algoritmo de Busca Alfa-Beta, Aprofundamento Iterativo e Tabela de Transposição 47

3 Estado da Arte **49**

3.1 Complexidade de Jogos 49

3.2 Jogadores Automáticos de Damas que utilizam Redes Neurais com Aprendizado por Reforço e Método das Diferenças Temporais 50

3.2.1 Jogadores com Forte Interferência Humana no Processo de Aprendizagem . . . 52
3.2.2 Jogadores sem Interferência Humana no Processo de Aprendizagem 54
3.3 Anaconda e a Computação Evolutiva . . . 57
3.4 Outros Jogos que Usam Aprendizagem por Reforço e Diferenças Temporais 58
3.4.1 O sucesso do *TD-Gammon* para o jogo de Gamão . . . 58
3.4.2 Xadrez . . . 59
3.5 Redes Neurais Auto-Organizáveis - *Kohonen-SOM* . . . 60

4 *VisionDraughts* - Um Sistema de Aprendizado de Damas com Eficiente Método de Busca 63
4.1 Representação do Tabuleiro nos Jogadores de Damas . . . 64
4.1.1 Representação Vetorial . . . 64
4.1.2 Representação NET-FEATUREMAP . . . 65
4.2 Arquitetura Geral do *VisionDraughts* . . . 69
4.3 Cálculo da Predição e Escolha da Melhor Ação . . . 71
4.4 Reajuste de Pesos da Rede Neural MLP . . . 73
4.5 Estratégia de Treino por *Self-Play* com Clonagem . . . 76
4.6 Eficiente Método de Busca do *VisionDraughts* . . . 77
4.6.1 O algoritmo Alfa-Beta . . . 78
4.6.2 A Tabela de Transposição . . . 84
4.6.3 Integração entre o Algoritmo Alfa-Beta e a Tabela de Transposição 93
4.6.4 O Aprofundamento Iterativo . . . 103
4.6.5 O Algoritmo Alfa-Beta com a Tabela de Transposição e o Aprofundamento Iterativo . . . 104
4.7 Análise em Retrocesso e Base de Dados . . . 106

5 *MP-Draughts* - Um Sistema Muti-Agente de Aprendizado de Damas 111
5.1 Ordenação da árvore de busca do *VisionDraughts* . . . 113
5.2 Arquitetura Geral do *MP-Draughts* . . . 118
5.3 ***IIGA*** - *Initial/Intermidiate Game Agent* . . . 120
5.4 ***EGA*** - *EndGame Agent* . . . 123
5.4.1 Obtenção de Base de Dados de Treinamento dos Agentes de Final de Jogos . . . 124
5.4.2 *Clusterização* da Base de Dados usando Redes de *Kohonen-SOM* 126
5.4.3 Treinamento das Redes Multiagentes de Final de Jogo . . . 132
5.4.4 Escolha do *EGA* . . . 133
5.5 Resultados Experimentais . . . 134
5.5.1 Impacto Causado no Tempo de Treinamento . . . 135

5.5.2 Impacto da Criação de Agentes Especializados em Fases Distintas do Jogo de Damas 136
5.5.3 Impacto da redução dos casos de *loop* 137

6 Conclusões e Propostas Futuras **139**
6.1 Perspectiva de Trabalhos Futuros 140

Referências Bibliográficas **141**

Lista de Figuras

2.1 Etapas do processo de *Clusterização* 32
2.2 Modelo de um neurônio artificial [46] 36
2.3 Modelo de uma função de ativação baseada na tangente hiperbólica 37
2.4 Arquitetura de um *Perceptron* Simples 39
2.5 Arquitetura de um *Perceptron* Multicamadas - MLP 40
2.6 Função Chapéu Mexicano descrevendo a ativação lateral do cérebro humano 41
2.7 Sensibilidade à orientação *versus* distância - Adaptado de Hubel e Wiesel (1962) 42
2.8 Modelo arquitetura de uma rede *Kohonen-SOM* 43
2.9 Exemplo de uma árvore de busca criada pelo minimax 46
2.10 Exemplo de uma árvore de busca criada pelo algoritmo alfa-beta 46

3.1 Exemplo de um tabuleiro do jogo de Gamão 58

4.1 Exemplo de tabuleiro representado por NET-FEATUREMAP 69
4.2 Fluxo de aprendizado do *VisionDraughts*: um sistema de aprendizagem de jogos de Damas 70
4.3 Rede Neural utilizada pelo *VisionDraughts* 72
4.4 A esquerda: árvore de busca expandida pelo minimax. A direita: árvore expandida pelo Alfa-Beta. 83
4.5 Exemplo de transposição em c e f : o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com peças simples. 84
4.6 Exemplo de transposição em a e c: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com reis. 85
4.7 Vetor de 128 elementos inteiros aleatórios utilizados pelo *VisionDraughts* . 87
4.8 Exemplo de movimento simples. 88
4.9 Árvore de busca onde o pai de n é maximizador 98
4.10 Árvore de busca onde o pai de n é minimizador 99
4.11 Árvore de busca criada pelo Aprofundamento Iterativo com limite de profundidade 3 104

5.1 Exemplo de ordenação da árvore de busca com aprofundamento iterativo . 114
5.2 Arquitetura geral do *MP-Draughts*: um sistema multiagente de aprendizagem de jogos de Damas. 119
5.3 Fluxo de aprendizado do *IIGA*: um sistema de aprendizagem para fases iníciais/intermediárias de jogos de Damas 122
5.4 Espaço de estados para o jogo de Damas: quantidade de estados possíveis de acordo com o número de peças sobre o tabuleiro (SCHAEFFER et al., 2007). 125
5.5 Arquitetura do processo de *Clusterização* 127

Lista de Tabelas

3.1 Complexidade do espaço de estados e fator de ramificação de alguns jogos de tabuleiro 50

4.1 Conjunto de 28 Características implementadas por Samuel 68
4.2 Conjunto de Características implementadas no jogador *VisionDraughts* . . 70

5.1 Conjunto de Características implementadas no jogador *MP-Draughts* . . . 121
5.2 Tempo de treinamento dos jogadores *VisionDraughts* e *MP-Draughts* . . . 135
5.3 Resultado de torneio de 20 jogos entre: *MP-Draughts*, *VisionDraughts* e *NeuroDraughts* 136
5.4 Número de empates causados por *loop* em 20 jogos entre os jogadores *MP-Draughts*, *VisionDraughts* e *NeuroDraughts* 137

Capítulo 1

Introdução

1.1 Introdução e Motivação

A escolha do jogo de Damas como domínio de verificação da eficiência das técnicas da Inteligência Artificial deve-se ao fato de que ele apresenta significativas semelhanças com inúmeros problemas práticos do dia-a-dia e, por outro lado, apresenta uma complexidade que demanda a utilização dos recursos propiciados por técnicas poderosas, tais como: *clusterização* de dados, método das Diferenças Temporais TD(λ), busca minimax, poda alfa-beta, aprofundamento iterativo, tabela de transposição, redes neurais e algoritmos genéticos.

Como exemplos de problemas práticos referentes ao o uso de técnicas inteligentes para serem resolvidos, podem-se citar:

- *Problema de navegação em que os mapas são obtidos autonomamente por um robô móvel*: a tarefa de aprendizagem parte de um ponto de referência inicial, em que o robô deve aprender uma trajetória de navegação de modo a atingir um ponto alvo e ao mesmo tempo, desviar-se dos obstáculos que surgem no ambiente [59].
- *Problema de interação com humanos por meio de diálogo:* a vida moderna, cada vez mais, precisa de agentes que dialoguem com seres humanos; um exemplo disso são os atendentes eletrônicos em empresas de prestação de serviços. Como exemplo de sistema que ataca esse problema, cita-se o sistema ELVIS - *Elvis Voice Interactive System* - de Walker [83]. Este sistema cria um agente que aprende a escolher uma boa estratégia de diálogo com humanos por meio de suas experiências e interações com usuários humanos.
- *Problema do controle de tráfego veicular urbano:* o objetivo é criar um agente capaz de controlar o número médio de veículos sobre uma rede urbana de forma a minimizar os congestionamentos e o tempo de viagem [84].

Note que os problemas práticos apresentados anteriormente apresentam dificuldades

similares aos problemas encontrado no domínio dos jogos de Damas como, por exemplo:

- aprender a se comportar em um ambiente em que o conhecimento adquirido é armazenado em uma função de avaliação;
- escolher um mínimo de atributos possíveis que melhor caracterizem o domínio e que sirvam como um meio pelo qual a função de avaliação adquirirá novos conhecimentos (esta questão é fundamental para se obter agentes com alto nível de desempenho);
- selecionar a melhor ação para um determinado estado ou configuração do ambiente onde o agente está inserido (problema de otimização).

Diante do vasto campo de pesquisa proporcionado pelo domínio dos jogos de Damas, Mark Lynch desenvolveu um jogador automático, *NeuroDraughts*, que utiliza técnica de aprendizagem por Diferenças Temporais para ajustar os pesos de um *perceptron* multicamadas cujo papel é estimar o quanto um estado de tabuleiro do jogo, representado em sua camada de entrada por um conjunto de características específicas do próprio jogo, é favorável ao agente jogador [42], [43].

Como o *NeuroDraughts* utilizou um conjunto de características definido manualmente, Neto e Julia desenvolveram o *LS-Draughts* [50], [51], [52], um sistema que expande o trabalho de Lynch por meio de técnica de algoritmos genéticos, gerando automaticamente um conjunto mínimo e essencial de características para representar o jogo de Damas. Esse conjunto mínimo de características encontradas por Neto e Julia otimizou o treinamento e o desempenho do jogador automático de Mark Lynch.

Continuando as pesquisas, Caixeta e Julia desenvolveram o *VisionDraughts*. Esse sistema substitui o módulo de busca minimax do *NeuroDraughts* por uma rotina de busca em árvore eficiente baseada no algoritmo alfa-beta com tabela de transposição e aprofundamento iterativo, tornando o processo de treinamento do jogador extremamente eficiente e, consequentemente, aumentando seu desempenho. Porém, no que diz respeito a bom desempenho, o *VisionDraughts* e outros jogadores de Damas, como: Anaconda [14], [19], *NeuroDraughts* [42], [43] e *LS-Draughts* [50], [51], [52] que aprendem por reforço, frequentemente, não terminam a fase final de um jogo com sucesso, mesmo estando em situação de vantagem quando comparado com seu oponente (problema de *loop* de final de jogo).

Pretendendo atacar tal problema de finalização de jogo existente nesses jogadores, o *MP-Draughts* estende e aprimora o *VisionDraughts* por meio de uma arquitetura multiagente em que cada um dos agentes que compõem o sistema é especializado em uma fase distinta do jogo, conforme sugestão dos próprios autores do *LS-Draughts* [50], [51], [52] e *VisionDraughts* [12], [11].

O projeto do *MP-Draughts* teve por objetivo gerar um jogador multiagente de Damas competitivo e com o mínimo possível de intervenção humana usando somente eficientes

técnicas da Inteligência Artificial para seu aprendizado. O sistema conta com 26 agentes jogadores, sendo um agente especializado nas fases iniciais e intermediárias de jogo (caracterizadas por estados de tabuleiro com, no mínimo, 13 peças); e os outros 25 agentes especializados em fase de final de jogo (caracterizada por tabuleiros com, no máximo, 12 peças). Cada um dos agentes que compõe o *MP-Draughts* apresenta uma arquitetura aprimorada do *VisionDraughts*, ou seja, uma rede neural multicamadas que usa Diferenças Temporais TD(λ), algoritmo alfa-beta com tabela de transposição e aprofundamento iterativo para aprender a jogar com o mínimo possível de interferência humana. Contudo, conforme será apresentado, a arquitetura dos agentes do *MP-Draughts* é mais otimizada do que a do *VisionDraughts*, uma vez que conta com o procedimento adicional de ordenação parcial da árvore de busca que agiliza, consideravelmente, o processo de escolha do melhor movimento pelo agente. Dos 25 agentes especializados em final de jogo presentes na estrutura do *MP-Draughts*, cada um é treinado para ser capaz de lidar com um determinado *cluster* de estados de tabuleiro de final de jogo diferente. Esses 25 *clusters* são minerados de um base de dados com 4000 estados de tabuleiro de final de jogo por meio de um algoritmo de redes de *Kohonen-SOM* [35], [34], [37], [38].

Realizaram-se alguns jogos envolvendo os jogadores *NeuroDraughts*, *VisionDraughts* e *MP-Draughts* e os resultados obtidos indicam que, com o uso de um jogador multiagente, o número de ocorrências do problema do *loop* reduziu; além disso, o desempenho do jogador aumentou consideravelmente.

Convém salientar que os bons resultados obtidos com o *MP-Draughts* permitiram a publicação do artigo "*MP-Draughts: a Multiagent Reinforcement Learning System based on MLP and Kohonen-SOM Neural Networks*"na conferência "2009 *IEEE International Conference on Systems, Man, and Cybernetics*"(SMC2009).

1.2 Estrutura da dissertação

Os próximos capítulos deste trabalho estão organizados conforme disposto a seguir:

Capítulo 2 : apresentação do referencial teórico que aborda as importantes técnicas da Inteligência Artificial utilizadas no desenvolvimento do agente multiagente *MP-Draughts*, que são: agentes inteligentes multiagentes, *clusterização* de dados, redes neurais, aprendizagem por reforço, treinamento por Diferenças Temporais e estratégia de busca.

Capítulo 3 : estado da arte de sistemas que utilizam das técnicas inteligentes apresentadas no capítulo sobre referencial teórico para desenvolver agentes inteligentes capazes de executar bem a tarefa para a qual foram projetados. Os sistemas abordados nesse capítulo focam, principalmente, o ambiente de jogos de tabuleiros e, mais especificamente, o domínio de jogo de Damas.

Capítulo 4 : apresentação do agente jogador *VisionDraughts*, no qual o *MP-Draughts* foi inspirado. O *VisionDraughts* é um sistema de aprendizagem de jogos de Damas baseado em redes neurais, Diferenças Temporais, algoritmos de busca em árvores e informações perfeitas contidas em bases de dados;

Capítulo 5 Apresentação do projeto e do desenvolvimento do sistema *MP-Draughts*: um sistema multiagente jogador de Damas baseado em *clusterização* de dados, redes neurais, Diferenças Temporais e algoritmos de busca em árvores. Nesse capítulo apresentam-se também os bons resultados obtidos por tal sistema.

Capítulo 6 : conclusões e perspectivas de trabalhos futuros.

Capítulo 2

Referencial Teórico

Neste capítulo, apresentam-se as técnicas usadas no desenvolvimento do agente jogador de Damas, *MP-Draughts*. As técnicas, aqui mostradas, são usadas no desenvolvimento de vários outros jogadores automáticos, desde jogadores de Damas a jogadores de Gamão, Xadrez etc.

A técnica de Sistemas Multiagentes foi usada para criar um jogador de Damas multiagente que possui jogadores especializados em fases distintas do jogo. Usou-se a técnica de *Clusterização* Redes *Kohonen-SOM* [37], [38] para agrupar os estados de tabuleiros de final de jogo nos quais o jogador foi treinado. As Redes Neurais são usadas no processo de *clusterização* dos estados de tabuleiro, como citado acima, e também no processo de aprendizagem do jogador combinada com Aprendizagem por Reforço e Método das Diferenças Temporais. A eficiente estratégia de busca apresentada foi usada para buscar, na árvore de busca do jogo, o melhor movimento que o agente jogador deve executar em um dado instante do jogo.

2.1 Sistemas Multiagentes

Adotar a abordagem de multiagentes para a resolução de problemas tem sido um assunto muito explorado nas últimas duas décadas. Inúmeros trabalhos têm apresentado conceitualizações, formalizações, protocolos, técnicas e métodos para aplicação desse tipo de abordagem na concepção de *software*. Isso tem acontecido pelo fato de a abordagem multiagente possuir algumas características que viabilizam a resolução de problemas de outra forma que não a tradicional, adequando-se a problemas complexos [31], [29] e de natureza descentralizada [3].

Esse paradigma adota o conceito de agente para caracterizar uma unidade autônoma de resolução de problemas [44]. A partir disso, cria-se uma solução por agrupamento de agentes que trabalham cooperativamente, cada um deles resolvendo uma parte do problema. A esse agrupamento dá-se o nome de Sistema Multiagente (SMA). Segundo Jennings [30], SMA também refere-se à subárea da Inteligência Artificial Distribuída (IAD)

que investiga o comportamento de um conjunto de agentes autônomos, objetivando a solução de um problema que está além das capacidades de um único agente.

2.1.1 Sistemas Multiagentes e o Conceito de Agente

Um agente é uma entidade de *software* que exibe um comportamento autônomo e pró-ativo orientado a objetivos. Um agente está situado em algum ambiente sobre o qual é capaz de realizar ações para alcançar seus próprios objetivos de projeto e a partir do qual percebe alterações [44].

Wooldridge [44] visualiza um agente como sendo uma entidade com capacidade de resolução de problemas encapsulados. Inserido nessa visão, define-se o agente como tendo as seguintes propriedades:

- **autonomia** [1]: executam a maior parte de suas ações sem interferência direta de agentes humanos ou de outros agentes computacionais, possuindo controle total sobre suas ações e estado interno;
- **habilidade social**: por impossibilidade de resolução de certos problemas ou por outro tipo de conveniência, interagem com outros agentes (humanos ou computacionais), para completarem a resolução de seus problemas, ou ainda para auxiliarem outros agentes. Disso surge a necessidade de que os agentes tenham capacidade para comunicar seus requisitos aos outros e um mecanismo decisório interno que defina quando e quais interações são apropriadas;
- **capacidade de reação**: percebem e reagem à alterações no ambientes em que estiverem inseridos;
- **capacidade pró-ativa**: os agentes apresentam um comportamento orientado a objetivos, tomando iniciativas quando julgarem apropriado e, além disso, atuam em resposta às alterações ocorridas em seu ambiente.

Como um agente é uma entidade que encapsula conhecimento sobre algum domínio, nada mais natural do que agrupar agentes que possuam parte do conhecimento envolvido na estratégia de resolução de um problema e que, a partir disso, interajam com o objetivo de complementarem suas habilidades (ideia usada para o desenvolvimento do jogador multiagente). Assim, da mesma forma que, no mundo real, existem empresas com funcionários possuidores de diferentes habilidades e que, utilizando essas habilidades, desenvolvem parte das atividades necessárias ao processo produtivo, pode-se compor uma sociedade de agentes em que para cada agente seja alocada um subconjunto das habilidades requeridas pela estratégia de solução, em que a parte das tarefas a serem cumpridas sejam designadas

[1] Para ter autonomia, o agente deve ter um certo grau de inteligência. Isso o capacita a sobreviver em um ambiente dinâmico e por vezes ameaçador. Entende-se autonomia como a capacidade de o agente agir por seus próprios objetivos sem a intervenção de outrém.

a cada um. De acordo com sua disponibilidade de recursos (computacionais, materiais, tempo, conhecimento etc.).

Pode-se distinguir duas principais classes de sistemas multiagentes [17]:

- **Sistemas de Resolução Distribuída de Problemas**: nesses sistemas os agentes envolvidos são explicitamente projetados para, de maneira cooperativa, atingirem um dado objetivo, considerando-se que todos eles são conhecidos, à priori, e supondo que todos são benevolentes, existindo, desta forma, confiança mútua em relação às suas interações;
- **Sistemas Abertos**: nesses sistemas, os agentes não são necessariamente projetados para atingirem um objetivo comum, podendo ingressar e sair do sistema de maneira dinâmica. Nesse caso, a dinâmica de agentes desconhecidos precisa ser levada em consideração, bem como a possível existência de comportamento não benevolente no curso das interações.

Dentro da segunda classificação, estão inseridos os Sistemas Multiagentes. Nesse tipo de sistema, investiga-se o comportamento de um conjunto de agentes autônomos, possivelmente, pré-existentes, que interagem objetivando a resolução de um problema que está além das capacidades de um único indivíduo. Dessa forma, o comportamento global do sistema deriva da interação entre os agentes que fazem parte dele [17]. A partir disso, está envolvida a busca por uma funcionalidade nesse sistema que permita a esses agentes a capacidade de coordenar seus conhecimentos, objetivos, habilidades e planos individuais de uma forma conjunta, em favor da execução de uma ação ou da resolução de algum problema onde se faça necessária a cooperação entre os agentes.Nesses casos, diz-se que o agente exibe um comportamento social [17].

Moulin e Chaib-Draa [3] evidenciam as características que constituem vantagens significativas dos Sistemas Multiagente sobre um Sistema de único agente, entre elas: maior eficiência na resolução de problemas; - mais flexibilidade por possuir agentes de diferentes habilidades agrupados para resolver problemas; e aumento da segurança pela possibilidade de agentes assumirem responsabilidades de agentes que falham.

2.2 *Clusterização*

Clusterização é a divisão de dados, com base na similaridade entre eles, em grupos disjuntos chamados *clusters*. Isso significa que dados, em um mesmo *cluster*, são mais similares do que dados pertencentes a outros *clusters*. O ato de agrupar dados pode ser definido como um problema de aprendizado não-supervisionado, já que a estrutura dos dados e as propriedades que os tornam similares são desconhecidas. Como não existem

Figura 2.1: Etapas do processo de *Clusterização*

rótulos iniciais, o objetivo da *clusterização* é encontrar uma organização válida e conveniente dos dados, ao invés de separá-los em categorias como acontece no reconhecimento de padrões e na classificação de dados [1].

O processo de *clusterização* é dividido em 4 etapas. Essas etapas são ilustradas na figura 2.1. A seguir, são apresentadas cada uma dessas etapas que compõem o processo de *clusterização*:

1. *seleção de variáveis:* etapa em que são identificadas as variáveis, ou atributos, mais relevantes do conjunto de dados inicial;
2. *medidas de similaridade:* para que a proximidade de dois dados possa ser quantificada, é necessário adotar alguma medida de similaridade entre eles. Existem diversas maneiras de quantificar a similaridade entre pares de dados, e a escolha da medida de similaridade adequada é fundamental para a *clusterização* dos dados. A maneira mais comum de calcular a similaridade é usando distância Euclidiana. Maiores detalhes sobre medida de similaridade serão apresentados na seção 2.2.1;
3. *algoritmos de* clusterização*:* nessa etapa define-se o modo de agrupamento dos dados, que pode ser realizado de diferentes maneiras. Os algoritmos de *clusterização* classificam-se de acordo com as diferentes técnicas empregadas por eles no agrupamento dos dados. Alguns dos mais famosos algoritmos de *clusterização* serão apresentados na seção de técnicas de *clusterização* (seção 2.2.2). Nessa seção,apresentar-se-á também o algoritmo de redes de *Kohonen-SOM* usado, neste trabalho, na fase de *clusterização* dos estados de tabuleiros de final de jogo;
4. *validação e análise dos resultados:* nessa etapa avalia-se a qualidade dos *clusters* encontrados, já que são desconhecidos inicialmente. A análise dos resultados pode levar à redefinição dos atributos escolhidos, da medida de similaridade ou algoritmos de *clusterização* definidos nas etapas anteriores.

As 4 etapas do processo de *clusterização* apresentadas anteriormente são essenciais para se obter bons conjuntos de dados. Entre essas 4 etapas, as de maior peso para se ter bons conjuntos de dados são medidas de similaridade e os algoritmos de *clusterização*. Pensando nisso, as próximas subseções dedicam-se a clarificar o entendimento sobre essas duas fases.

2.2.1 Medida de Similaridade

Muitos métodos de *clusterização* utilizam, como ponto de partida, um vetor que reflete, de maneira quantitativa, a proximidade entre os elementos de um conjunto de dados. Essa proximidade pode representar a distância ou similaridade entre dois elementos. Quanto maior a similaridade, ou menor a distância entre dois elementos, mais próximos esses elementos encontram-se dentro de um conjunto [4]. Esse vetor recebe o nome de vetor de similaridade ou vetor de proximidade.

As medidas de similaridade variam de acordo com a representação e escala dos atributos dos dados. Um atributo pode ter representação binária, discreta ou contínua. A escala indica o grau de importância de um atributo em relação aos demais. Por exemplo, um atributo pode indicar a porcentagem de aceitação de um produto no mercado (onde o valor do atributo varia de 0 a 100), ou indicar o peso de uma determinada característica em um tabuleiro de Damas (seção 4.1.2), onde o valor absoluto do atributo é relevante.

Um vetor de similaridade representa a proximidade entre dados de um conjunto inicial. Uma maneira de medir a similaridade entre dados é por meio do cálculo da distância entre eles. Entre os métodos de cálculo de distância , os mais conhecidos são: distância *Euclidiana*, distância de *Manhattan* e distância "*Sup*", todas variações da distância de *Minkowski* [28].

A distância de *Minkowski* é denominada pela equação:

$$d(xi, xj) = \sqrt[p]{\sum_{k=1}^{d}(|x_{ik} - x_{jk}|)^2}, \text{ p} \geq 1 \ ,$$

onde: *d(xi, xj)* denota a proximidade entre os dados *xi* e *xj*, e o valor de p é que define qual das variações, citadas acima, deve ser usada. Quando $p = 1$, usa-se distância de *Manhattan*, se $p = 2$, usa-se distância Euclidiana e, se $p \rightarrow \infty$, usa-se distância "Sup".

Para se decidir qual métrica usar para calcular a distância entre dois dados, deve-se observar os tipos de dados que serão agrupados. A variação do parâmetro p define distâncias diferentes. Por exemplo, para dados de valores contínuos, o cálculo da distância Euclidiana é mais apropriado, ou seja, $p = 2$.

Como os dados a serem agrupados no decorrer deste trabalho, são dados de valores contínuos, o cálculo de proximidade usado foi distância Euclidiana. Detalhes de como foram feitos os cálculos são apresentados na seção 5.4.2 deste trabalho.

2.2.2 Técnicas de *Clusterização*

Na seção 2.2.1 relacionaram-se algumas medidas para quantificar a similaridade entre dados. O vetor de similaridade que representa essa proximidade é a entrada para os algoritmos de *clusterização*. Os algoritmos utilizados no processo de *clusterização* de dados podem ser classificados, por exemplo, de acordo com a abordagem utilizada na definição dos clusters: particionamento, redes auto-organizáveis, baseado em densidade

e algoritmos hierárquicos. As características desses algoritmos são apresentadas, brevemente, nas subseções a seguir, dando destaque aos algoritmos de *clusterização* por redes auto-organizáveis, que foram usados nesse trabalho.

Algoritmos de *clusterização* por particionamento

Na clusterização *por particionamento*, o conjunto de dados é dividido em um número determinado de clusters uma única vez. O fato de gerar a partição uma única vez passa a ser uma vantagem, quando a quantidade de dados é muito grande e a construção e armazenamento de todas as possibilidades de divisão resultantes da *clusterização* hierárquica tornam-se muito custosas [53].

O problema de *clusterização* por particionamento pode ser definido formalmente como: Dado um conjunto de n dados caracterizados por d atributos cada, determine uma partição do conjunto inicial em K *clusters*. A escolha do valor de K depende do problema abordado e pode interferir na eficiência do algoritmo. O objetivo é maximizar a similaridade entre elementos de um mesmo *cluster* e minimizar a similaridade entre elementos de *clusters* diferentes [28].

Algoritmos de *clusterização* por densidade

Um dado com d atributos pode ser representado como um ponto em um espaço d-dimensional e os *clusters* correspondem a subconjuntos de dados que estejam próximos. Dessa forma, os *clusters* localizam-se em regiões de maior densidade no espaço de métricas e são separados por regiões de baixa densidade. Os métodos de *clusterização* por densidade utilizam critério de *clusterização* local, por considerarem a densidade de ligações entre os dados. A possibilidade de encontrar *clusters* de formas arbitrárias e o fato de não precisar da definição do número de *clusters* [2] como parâmetro inicial são as principais vantagens dos algoritmos baseados em densidade.

Algoritmos de *clusterização* hierárquica

Nessa abordagem, são produzidas diversas partições do conjunto de dados com base na junção ou divisão dos *clusters* de acordo com a medida de similaridade. Conforme o modo de produção das partições, os métodos de *clusterização* hierárquicos podem ser classificados como *aglomerativos* ou *divisórios* [53].

Em algoritmos aglomerativos, no passo inicial, cada elemento do banco de dados forma um *cluster*. Nas próximas iterações, pares de *clusters* da iteração precedente que satisfazem um certo critério de distância mínima são aglutinados em um único *cluster*. O processo termina quando um número de *clusters* k fornecido pelo usuário é atingido [4].

Em algoritmos divisórios, no passo inicial, cria-se um único *cluster* composto pelo banco de dados inteiro. Nas próximas iterações, os *clusters* são subdivididos em duas

partes de acordo com algum critério de similaridade. O processo termina quando se atinge um número de *clusters k* fornecido pelo usuário [4].

Redes Auto-Organizáveis - SOM

A aplicação mais comum de redes neurais é feita na classificação de dados, no qual dados são separados em grupos previamente conhecidos. Entretanto, um tipo particular de rede neural, a rede auto-organizável, ou *Kohonen-SOM* (*Self Organizing Maps*), desenvolvida por Teuvo Kohonen [35] [36] pode separar dados em grupos desconhecidos inicialmente por meio de aprendizado não-supervisionado [4].

Em linhas gerais, as redes auto-organizáveis são formadas por um conjunto de neurônios, onde cada dado tem seus atributos conectados a todos os neurônios da rede. A essa ligação entre neurônio e atributo é dado um peso, inicialmente, aleatório. O aprendizado ocorre à medida que os dados são apresentados à rede, e o neurônio com conjunto de pesos mais próximo do dado é escolhido para representá-lo. O neurônio escolhido tem seus pesos alterados a fim de representar melhor o dado atribuído a ele. Assim cada neurônio torna-se especialista na identificação dos atributos.

Redes *Kohonen-SOM* foi o algoritmo escolhido para o processo de *clusterização* deste trabalho. Escolheu-se esse algoritmo pelo fato de sua aprendizagem ser baseada em técnicas de Inteligência Artificial(o que respeita a ideia geral deste trabalho cuja proposta é usar técnicas de Inteligência Artificial para ensinar um agente automático a jogar Damas). Outro fato importante na escolha do algoritmo de*clusterização Kohonen-SOM* foi o conhecimento que os autores deste trabalho têm sobre ele.

Maiores detalhes sobre o algoritmo de *clusterização* por Redes *Kohonen-SOM* são apresentados na seção 2.3.4.

2.3 Redes Neurais

O trabalho em Redes Neurais Artificiais (RNAs), usualmente denominadas "redes neurais", tem sido motivado, desde o começo, pelo reconhecimento de que o cérebro humano processa informações de uma forma inteiramente diferente do computador digital convencional. O cérebro é um computador altamente complexo, não-linear e paralelo. Ele tem a capacidade de organizar seus neurônios de forma a realizar certos processamentos muito mais rapidamente que o mais rápido computador digital existente [25].

As subseções a seguir tratam os principais aspectos da abordagem de redes neurais, apresentando: a modelagem matemática do neurônio e os tipos de redes neurais.

Destaca-se, entretanto, que, em virtude da grande diversidade de arquiteturas encontradas na literatura, apenas as de maior importância, ou de alguma forma relevantes ao trabalho proposto serão abordadas, tais como: *Perceptron* Multicamada e Redes Auto-

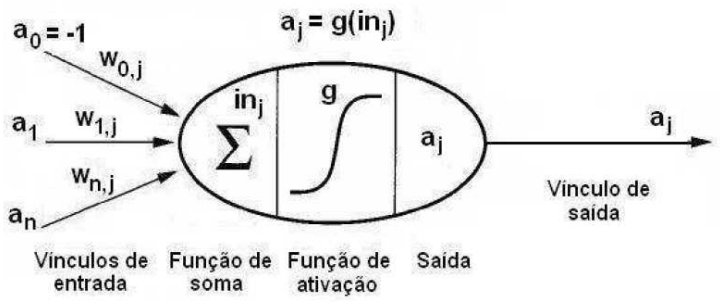

Figura 2.2: Modelo de um neurônio artificial [46]

Organizáveis *Kohonen-SOM*. Um panorama geral de outras arquiteturas pode ser encontrado em algumas referências bibliográficas da área [25], [18], [39].

2.3.1 A modelagem Matemática do Neurônio

Uma rede neural artificial é um modelo computacional, baseado em redes neurais biológicas, que consiste em uma rede de unidades básicas simples chamadas neurônios. Um mesmo algoritmo é executado em cada um dos neurônios e cada um deles se comunica com um subconjunto qualquer de outros neurônios da mesma rede [47].

O primeiro modelo matemático de um neurônio artificial foi proposto por McCulloch e Pitts em 1943 [46] e pode ser visto na figura 2.2. Nesse modelo, tem-se:

1. o neurônio é referenciado pela letra j;
2. os vínculos de entrada são representados por a_0, a_1 até a_n. Exceto a entrada a_0 que está fixa e vale -1, as demais entradas do neurônio j representam saídas de outros neurônios da rede;
3. os pesos são referenciados pela letra w. O peso w_{1j} , por exemplo, define o grau de importância que o vínculo de entrada a_1 possui em relação ao neurônio j;
4. a função de soma acumula os estímulos recebidos pelos vínculos de entrada a fim de que a função de ativação possa processá-los. A função de ativação é dada por $a_j = g(in_j) = g(\sum_{i=0}^{n} w_{ij}.a_i)$, onde a_i é a ativação de saída do neurônio i conectado a j, e w_{ij} é o peso na ligação entre os neurônios i e j;
5. o peso w_{0j} , conectado à entrada fixa a_0 = -1, define o limiar para o neurônio j, no sentido de que o neurônio j será ativado quando a soma ponderada das entradas reais $\sum_{i=1}^{n} w_{ij}.a_i$ exceder $w_{0j}.a_0$.

A função de ativação g, ou *camada de processamento de limiares*, é projetada para atender a duas aspirações:

1. o neurônio deverá ser ativado, se, e somente se, as entradas recebidas superarem o limiar do neurônio;

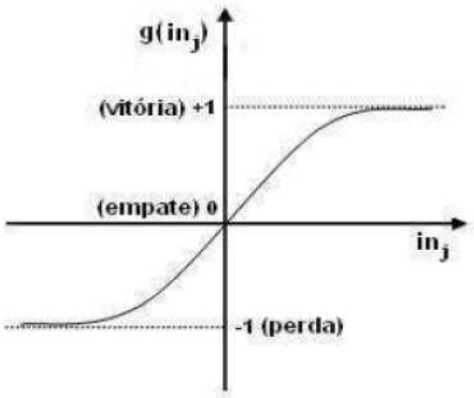

Figura 2.3: Modelo de uma função de ativação baseada na tangente hiperbólica

2. a ativação precisa ser não-linear para redes MLP, caso contrário, a rede neural se torna uma função linear simples;

Um exemplo de função de ativação é a tangente hiperbólica mostrada na figura 2.3.

O ajuste sináptico entre os neurônios de uma RNA representa o aprendizado, em cada neurônio, do fato apresentado, isto é, cada neurônio, conjuntamente com todos os outros, representa a informação que atravessou pela rede. Nenhum neurônio guarda em si todo o conhecimento, mas faz parte de uma malha que retém a informação graças a todos os seus neurônios. Dessa forma, o conhecimento dos neurônios e, consequentemente, da própria rede neural, reside nos pesos sinápticos.

Desse modo, pode-se dizer que as redes neurais artificiais têm sido desenvolvidas como generalizações de modelos matemáticos de cognição humana ou neurobiologia, assumindo que:

- o processamento da informação ocorre com o auxílio de vários elementos chamados *neurônios*;
- os sinais são propagados de um elemento a outro por meio de *conexões*;
- cada conexão possui um *peso* associado que, em uma rede neural típica, pondera o sinal transmitido;
- cada neurônio possui uma *função de ativação* (geralmente não-linear), cujo argumento é a soma ponderada dos sinais de entrada, que determina a saída do neurônio.

Os tipos de redes neurais, apresentadas na próxima seção, são classificados de acordo com sua estrutura e a forma de treinamento.

2.3.2 Tipos de Redes Neurais

A abordagem conexionista das RNAs abre um amplo leque de formas de conexão entre as unidades de processamento, isto é, os neurônios. Isso abrange o número de camadas

presentes na rede, a forma de conexão entre tais unidades, a forma de treinamento, as funções de ativação presentes em cada camada etc.

Graças à grande bibliografia disponível sobre o tema RNA, até mesmo sua classificação gera algumas discussões. Fausett [18], por exemplo, define arquitetura de uma rede neural como a disposição dos neurônios em camadas e as conexões entre as camadas. Em um sentido mais amplo, outros pesquisadores utilizam a notação arquitetura na denominação de todo um conjunto de características de uma rede, englobando sua forma de treinamento, finalidade etc.

Em [86], Duc Pham define dois critérios básicos para a classificação das RNAs:

1. quanto à estrutura;
2. quanto à forma de treinamento.

Estruturas das RNAs

As redes neurais se classificam, quanto á estrutura, em acíclicas ou cíclicas [50], [86]:

- redes acíclicas ou redes de alimentação direta (*feedforward*): a propagação do processamento neural é feita em camadas sucessivas, ou seja, neurônios dispostos em camadas terão seus sinais propagados sequencialmente da primeira à última camada de forma unidirecional. Um exemplo típico desse tipo de rede seria o *Perceptron* Simples ou o *Perceptron* multicamadas;
- redes cíclicas ou redes recorrentes (*recurrent*): as saídas de um (ou todos) os neurônios podem ser realimentadas a neurônios de camadas precedentes (tipicamente da primeira camada). Esse tipo de rede é classificada como memória dinâmica. Um exemplo típico dessa rede é a rede de Hopfield (HOPFIELD, 1982).

O Treinamento das RNAs

Haykin propõe a seguinte definição para o aprendizado no contexto de redes neurais: "(...) é um processo pelo qual os parâmetros livres de uma rede neural são adaptados por meio de um processo de estimulação pelo ambiente no qual a rede está inserida. O tipo de aprendizagem é determinado pela maneira pela qual a modificação dos parâmetros ocorre" [25].

As redes neurais se classificam, quanto à forma de treinamento, em redes com treinamento supervisionado, com treinamento não supervisionado e com aprendizagem por reforço [50], [86]:

- **redes com treinamento supervisionado**: uma sequência de padrões de entrada associados a padrões de saída é apresentada à rede; daí ela utiliza comparações entre a classificação para o padrão de entrada e a classificação correta do padrão de saída para ajustar seus pesos;

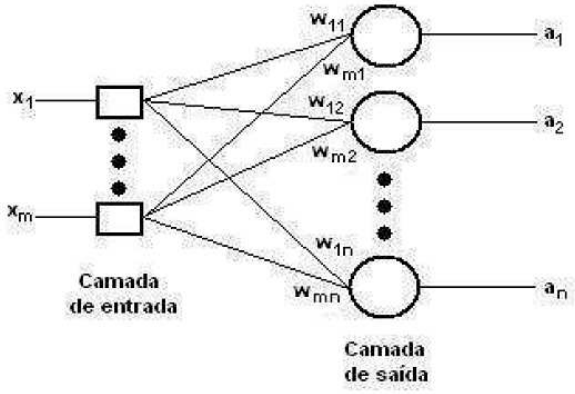

Figura 2.4: Arquitetura de um *Perceptron* Simples

- **redes com treinamento não supervisionado** não existe a apresentação de mapeamentos entrada-saída para a rede. Para esse tipo de treinamento não se usa um conjunto de exemplos previamente conhecidos. Uma medida da qualidade da representação do ambiente é estabelecida e os pesos da rede são modificados de modo a otimizar tal medida;
- **redes com aprendizagem por reforço**: utiliza-se alguma função heurística para descrever o quanto uma resposta da rede para uma determinada entrada é boa. Não se fornece à rede o mapeamento direto entrada-saída, mas sim uma recompensa (ou penalização) decorrente da saída gerada pela rede em consequência da entrada apresentada. Tal reforço é utilizado no ajuste dos pesos da rede.

O conhecimento adquirido e mantido por uma rede neural reside em seus pesos, porque nenhum neurônio guarda em si todo o conhecimento, mas faz parte de uma malha que retém a informação graças a todos os seus neurônios [50]. Ajustar os pesos de uma rede neural significa treiná-la com algum tipo de treinamento, seja supervisionado, não supervisionado ou com aprendizagem por reforço. Particularmente, a aprendizagem por reforço é essencial para o entendimento do sistema apresentado nesta dissertação e será, portanto, a única técnica de ajuste de pesos detalhada aqui (seção 2.4).

2.3.3 O *Perceptron* Simples e *Perceptron* Multicamadas - MLP

Os *perceptrons* de única camada são o tipo mais antigo de redes neurais, as quais são formadas por uma camada única de neurônios de saída que estão conectados às entradas $x_i(n)$ pelos pesos $w_{ij}(n)$, em que $x_i(n)$ representa o i-ésimo elemento do vetor padrão de entrada na iteração n; e, $w_{ij}(n)$ representa o peso sináptico conectando a entrada $x_i(n)$ à entrada do neurônio de saída j na iteração n, como apresentado na figura 2.4.

O modelo do *perceptron* de camada única de uma RNA consegue aprender apenas problemas linearmente separáveis. O *perceptron* simples é incapaz de resolver problemas

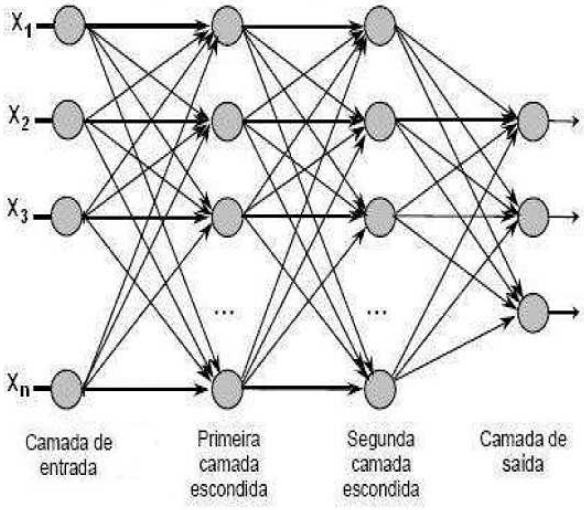

Figura 2.5: Arquitetura de um *Perceptron* Multicamadas - MLP

cujas funções não são linearmente separáveis, isto é, problemas que apresentam características de comportamento não linear.

Os *perceptrons* multicamadas ou MLPs se caracterizam pela presença de uma ou mais camadas intermediárias ou escondidas (camadas em que os neurônios são efetivamente unidades processadoras, mas não correspondem à camada de saída). Adicionando-se uma ou mais camadas intermediárias, aumenta-se o poder computacional de processamento não-linear e armazenagem da rede. Em uma única camada oculta, suficientemente grande, é possível representar, com exatidão, qualquer função contínua das entradas. O conjunto de saídas dos neurônios de cada camada da rede é utilizada como entrada para a camada seguinte. A figura 2.5 ilustra uma rede MLP com duas camadas ocultas.

2.3.4 Redes Neurais Auto-Organizáveis - SOM

Nesta seção, será estudado um dos mais populares algoritmos na categoria de aprendizado não-supervisionado, as redes neurais conhecidas como Mapas Auto-Organizados de Kohonen (Self-Organizing Map - SOM), ou simplesmente Redes *Kohonen-SOM* [37], [36], [38].

As redes *Kohonen-SOM* foram inicialmente desenvolvidas pelo Prof. Teuvo Kohonen [35], [34] e constituem um dos principais paradigmas na área de redes neurais artificiais. Tais redes foram inspiradas no modo pelo qual informações sensoriais são mapeadas no córtex cerebral.

O desenvolvimento de uma rede *Kohonen-SOM*, como modelo neural, foi motivado por uma característica do cérebro humano. Assim descrever-se-á, na próxima subseção a motivação biológica para sua criação e, na sequência, será apresentado sua arquitetura. Estas seções restringir-se-ão a aspectos fundamentais de uma rede *Kohonen-SOM*.

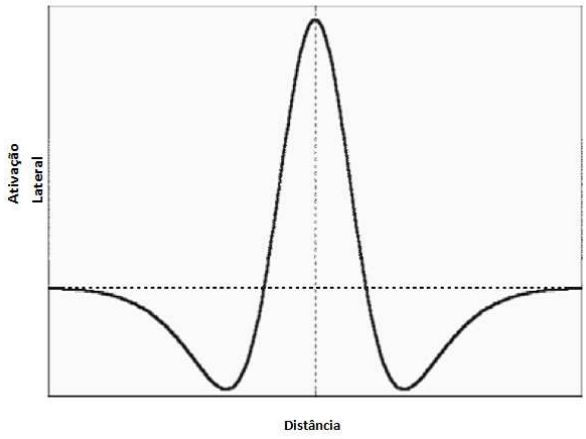

Figura 2.6: Função Chapéu Mexicano descrevendo a ativação lateral do cérebro humano

Motivação biológica

O córtex cerebral humano é uma fina camada de células de, aproximadamente, um metro quadrado, contendo seis camadas de neurônios redobradas para caber no crânio [21]. Apesar de a mecânica e os processos do córtex não serem ainda completamente entendidos, evidências anatômicas e fisiológicas sugerem a existência de iteração lateral entre os neurônios. No caso do cérebro dos mamíferos, ao redor de um centro de excitação existe uma região (50 a 100 μm) de excitação lateral. Ao redor dessa região, existe uma área de ativação inibitória, aproximadamente, de 200 a 500 μm [36]. Novamente, ao redor dessa ultima área, segue-se uma região de excitação fraca (alguns centímetros). O fenômeno de iteração lateral pode ser modelado pela função chapéu-mexicano (*Mexican hat*), mostrado na figura 2.6.

Atualmente sabe-se que o cérebro possui áreas especializadas para processar diferentes modalidades de sinais. Nessas áreas, principalmente, nas áreas dedicadas ao processamento primário de sinais sensoriais, os neurônios respondem às muitas qualidades dos estímulos de uma forma ordenada. Por exemplo, na área auditiva, existe uma "escala"para frequências acústicas diferentes. Na área visual, existem mapas para processar a orientação de segmentos de reta, mapas de cores [5] etc. Mapas ordenados topograficamente, geralmente, bidimensionais e o conceito de formação de imagens abstratas das dimensões das características sensoriais aparentam ser um dos mais importantes princípios na formação das representações internas no cérebro [36].

Talvez o trabalho experimental mais influente nessa área, tenham sido os estudos de Hubel e Wiesel [27] usando micro-eletrodos no córtex visual de gatos. Estímulos semelhantes, como segmentos de reta com orientação variando poucos graus, excitaram neurônios próximos. Registrando as respostas mais intensas com um eletrodo, inserido paralelamente à superfície do córtex e gradualmente movido ao longo do tecido nervoso, obtém-se uma série de orientações que, em geral, variam suavemente ao longo do córtex. A figura 2.7 mostra dados experimentais de Hubel e Wiesel, nos quais veem-se também

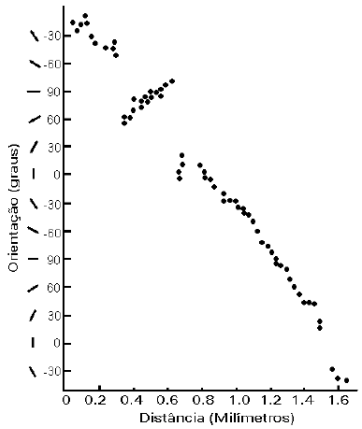

Figura 2.7: Sensibilidade à orientação *versus* distância - Adaptado de Hubel e Wiesel (1962)

descontinuidades ocasionais.

Percebe-se, na figura 2.7, uma espécie de mapeamento, onde orientações similares excitam regiões do tecido nervoso próximo. No caso do córtex auditivo, experimentos indicam uma escala logarítmica de frequências. Neurônios em posições diferentes são excitados por sons diferentes e as posições relativas refletem, de uma certa forma, o relacionamento entre o conjunto de sons [36].

Apesar de as ideias de auto-organização em sistemas neurais terem sido iniciadas no final dos anos 50 e início dos anos 60 [87], C. von der Malsburg demonstrou, pela primeira vez, em 1973, a possibilidade de treinar uma rede neural usando métodos competitivos de forma a criar um mapeamento semelhante ao apresentado na figura 2.7. Modelos de auto-organização biologicamente inspirados foram desenvolvidos [85] baseados no estudo de neurônios que respondem seletivamente a estímulos, como os sensíveis à intensidade de luz e orientação de segmentos de retas, no córtex visual.

Kohonen [35], [34] generalizou tal modelo na rede *Kohonen-SOM*. Previamente, Kohonen havia se dedicado a estudos sobre memória associativa e modelos para atividade neuro-biológica. A popularização da rede *Kohonen-SOM* deve-se a muitos fatores como, por exemplo, o seu esforço de manter uma fonte de referências sempre atualizada.

Estrutura básica de uma *Kohonen-SOM*

Apesar de ter sido desenvolvido inicialmente para tentar modelar áreas sensoriais do córtex e das iterações laterais entre os neurônios em tais estruturas, uma rede do tipo *Kohonen-SOM* tem um algoritmo extremamente simplificado e, apenas superficialmente, podemos compará-lo a uma estrutura biológica real do cérebro.

As redes *Kohonen-SOM* são compostas por duas camadas, sendo a primeira de entrada e a segunda de saída. A camada de entrada é por onde os dados a serem *clusterizados* são introduzidos na rede. Os nodos dessa camada são completamente conectados com os

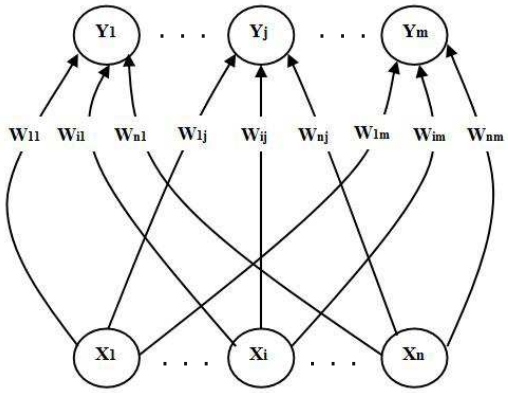

Figura 2.8: Modelo arquitetura de uma rede *Kohonen-SOM*

neurônios da segunda camada (camada de saída), sendo a segunda camada da rede competitiva. Em uma camada competitiva, os neurônios disputam entre si quem representará o dado apresentado na entrada de rede. A cada conexão entre um nodo da camada de entrada e um neurônio da camada de saída é associado um peso. A figura 2.8, mostra a arquitetura de uma rede *Kohonen-SOM*.

A arquitetura de uma rede *Kohonen-SOM*, mostrada na figura 2.8, inclui n nodos na camada de entrada e m neurônios na camada de saída, onde todos os nodos X_i da camada de entrada ($1 \leq i \leq n$) estão conectados a todos os neurônios Y_j da camada de saída ($1 \leq i \leq m$). O w_{ij} indica o peso da ligação entre o *i-ésimo* neurônio da camada de entrada e o *j-ésimo* neurônio da camada de saída. Adicionalmente o *vetor peso* que representa cada neurônio Y_j da camada de saída é composto de todos os n pesos w_{ij}, lembrando que ($1 \leq i \leq n$).

O algoritmo de *clusterização* da rede *Kohonen-SOM* será detalhado na seção 5.4.2.

2.4 Aprendizagem por Reforço

O paradigma da aprendizagem por reforço (AR) tem sido de grande interesse na área da aprendizagem automática, uma vez que dispensa um "professor"inteligente para o fornecimento de exemplos de treino, esse fato torna-o particularmente adequado a domínios complexos em que a obtenção desses exemplos seja difícil ou até mesmo impossível [60].

Segundo Sutton e Barto [77], na Aprendizagem por Reforço, um indivíduo deve aprender a partir da sua interação com o ambiente onde se encontra, pelo do conhecimento do seu próprio estado no ambiente, das ações efetuadas no ambiente e das mudanças de estado que aconteceram depois de efetuadas as ações.

A importância de utilizar AR como uma técnica de aprendizagem está diretamente ligada ao fato de se tentar obter uma política ótima de ações. Tal política é representada pelo comportamento que o agente segue para alcançar o objetivo e pela maximização de alguma medida de reforço a longo prazo (globais) nos casos em que não se conhece, a

priori, a função que modela esta política (função do agente-aprendiz).

Um sistema típico de aprendizagem por reforço constitui-se, basicamente, de um agente interagindo em um ambiente via percepção e ação. O agente percebe as situações dadas no ambiente (pelo menos parcialmente) e seleciona uma ação a ser executada em consequência de sua percepção. A ação executada muda, de alguma forma, o ambiente; e as mudanças são comunicadas ao agente por um sinal de reforço [50].

Formalmente, o modelo de um sistema de aprendizagem por reforço consiste em [77]:

- um conjunto de variáveis de estado percebidas por um agente. As combinações de valores dessas variáveis formam o conjunto de estados discretos do agente (S);
- um conjunto de ações discretas, que escolhidas pelo agente mudam o estado do ambiente (A(s), onde $s\epsilon$ S);
- um conjunto de valores das transições de estados (reforços tipicamente entre [0,1]).

O objetivo do método de aprendizagem por reforço é fazer com que o agente escolha uma sequência de ações que aumente a soma dos valores das transições de estados, ou seja, é encontrar uma política, definida como um mapeamento de estados em ações que maximize as medidas de reforço acumuladas ao longo do tempo.

Entre todos os algoritmos existentes para solucionar o problema da aprendizagem por reforço, este trabalho enfocará o algoritmo de Diferenças Temporais (TD(λ)) de Sutton, descrito com mais detalhe em [77].

2.4.1 Método de Solução por Diferenças Temporais

As Diferenças Temporais são capazes de utilizar o conhecimento prévio de ambientes parcialmente conhecidos para predizer o comportamento futuro (por exemplo, predizer se uma determinada disposição de peças no tabuleiro de Damas conduzirá à vitória).

Os métodos TD(λ) são guiados pelo erro ou diferença entre predições sucessivas temporárias de estados sequenciais experimentados por um agente em um domínio. Assim, o aprendizado do agente pelo método TD(λ) é extraído de forma incremental, diretamente da experiência desse agente sobre o domínio de atuação, atualizando as estimativas a cada passo, sem a necessidade de ter que alcançar o estado final de um episódio (um episódio pode ser definido como sendo um único estado ou uma sequência de estados de um domínio) (NETO, 2007).

O cálculo formal do reajuste dos pesos de uma rede neural, usando o método de Diferenças Temporais TD(λ) de Sutton [75] é detalhado, posteriormente, na seção 4.4, em que o mesmo é usado para reajuste dos pesos da rede neural jogadora.

2.5 Estratégia de Busca Eficiente

Os agentes inteligentes devem maximizar sua medida de desempenho para a resolução de um determinado problema. O processo de busca em um agente jogador é fundamental para a maximização do seu desempenho, porque é pelo processo de busca que o jogador encontra o melhor movimento a executar em um dado momento. Pensando nisso, foram usadas técnicas extremamente eficientes no processo de busca do jogador desenvolvido neste trabalho, as quais são apresentadas na sequência.

2.5.1 Algoritmo de Busca Alfa-Beta

De forma genérica, as estratégias de busca tradicionais envolvem uma busca em uma árvore que descreve todos os estados possíveis a partir de um estado inicial dado. Formalmente, o espaço de busca é constituído por um conjunto de nós conectados por arcos. A cada arco pode ou não estar associado um valor que corresponde ao custo c de transição de um nó a outro. A cada nó temos associada uma profundidade p, sendo que a mesma tem valor 0 no nó raiz e aumenta de uma unidade para um nó filho. A aridade a de um nó é a quantidade de filhos que o mesmo possui, e a aridade de uma árvore é definida como a maior aridade de qualquer um de seus nós. O objetivo da busca é encontrar um caminho (ótimo ou não) do estado inicial até um estado final explorando, sucessivamente, os nós conectados ao nós já explorados até a obtenção de uma solução para o problema.

Entretanto, nos problemas em que se deseja planejar, com antecedência, ações a serem executadas por um agente em um ambiente onde outros agentes estão fazendo planos contrários àquele, surge o chamado *problema de busca competitiva*. Nesses ambientes, as metas dos agentes são mutuamente exclusivas. Os jogos são exemplos de ambientes que apresentam este tipo de problema de busca competitiva: o jogador não tem que se preocupar apenas em chegar ao objetivo final, mas também em evitar que algum oponente chegue antes dele, ou seja, vença o jogo. Dessa maneira, o jogador deve se antecipar à jogada do seu adversário para poder fazer a sua jogada. Uma das maneiras de solucionar esse tipo de problema é utilizando o método de busca minimax.

O minimax [60] é uma técnica de busca para determinar a estratégia ótima em um cenário de jogo com dois jogadores. O objetivo dessa estratégia ótima é decidir a melhor jogada para um dado estado do jogo. Há dois jogadores no minimax: o *MAX* e o *MIN*. Uma busca em profundidade é feita a partir de uma árvore onde a raiz é a posição corrente do jogo. As folhas dessa árvore são avaliadas pela ótica do jogador *MAX*, e os valores dos nós internos são atribuídos de baixo para cima com essas avaliações. As folhas do nível minimizar são preenchidas com o menor valor de todos os seus nós filhos, e o nível de maximizar são preenchidos com o maior valor de todos os nós filhos. Como a quantidade de busca cresce exponencialmente com o aumento da profundidade de análise do algoritmo minimax, é necessário, para reduzir o tempo de busca, que nenhum tempo seja perdido

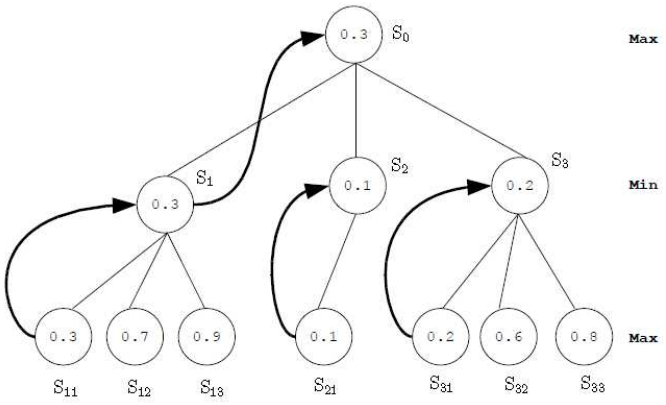

Figura 2.9: Exemplo de uma árvore de busca criada pelo minimax

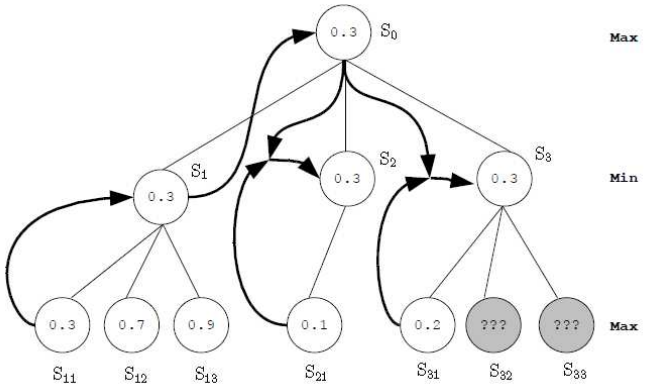

Figura 2.10: Exemplo de uma árvore de busca criada pelo algoritmo alfa-beta

analisando jogadas que obviamente não serão boas para o jogador. Um modo significativo de se conseguir isso é utilizando de poda alfa-beta no algoritmo minimax.

A figura 2.9 mostra um exemplo de aplicação do algoritmo minimax que gera a árvore de busca do jogo para um determinado estado.

O algoritmo minimax examina mais nós da árvore de busca do que o necessário e, nesse sentido, o mecanismo de poda alfa-beta elimina seções da árvore de busca que, definitivamente, não podem conter a melhor ação a ser executada pelo agente jogador. O algoritmo alfa-beta pode ser resumido como um procedimento recursivo que escolhe a melhor ação a ser executada fazendo uma busca em profundidade, da esquerda para a direita, em uma árvore de busca [72].

A figura 2.10 exemplifica como acontece uma poda alfa-beta em uma árvore de busca. Comparando as figuras 2.9 e 2.10, é possível verificar o quanto um mecanismo de poda pode economizar no tempo de busca pela melhor ação. Com essa economia no tempo de busca, é possível aumentar o nível de busca na árvore e consequentemente aumentar a chance de se encontrar melhores resultados para a busca.

A fim de proporcionar ainda mais eficiência ao mecanismo de busca, a combinação do algoritmo alfa-beta com aprofundamento iterativo e tabelas de transposição pode fornecer

bons resultados. Veja como funciona cada uma dessas técnicas nas seções 2.5.2 e 2.5.3, respectivamente.

2.5.2 Algoritmo de Busca Alfa-Beta e Aprofundamento Iterativo

A busca com aprofundamento iterativo combina os benefícios dos procedimentos de busca em largura e busca em profundidade [60]. Como na busca em largura, ela é completa quando o fator de ramificação é finito e ótima quando o custo do caminho é proporcional à profundidade. Como na busca em profundidade, ela economiza recursos computacionais enquanto tenta encontrar a solução do problema.

Uma busca com aprofundamento iterativo (*iterative deepening*) pode ser utilizada para tentar encontrar o limite de profundidade mais adequado que um algoritmo de busca deve descer em busca da melhor ação, levando em consideração as restrições de recursos computacionais e tempo de busca. Ela faz isso incrementando, gradualmente, o limite de profundidade até encontrar o estado objetivo mais raso, ou até atingir o limite de tempo fornecido para a busca.

A ideia básica da combinação do algoritmo de busca alfa-beta com o aprofundamento iterativo é realizar uma série de buscas, em profundidade, independentes, cada uma com um *look-ahead* (nível de profundidade de busca) acrescido de um nível. Assim, é garantido que o procedimento de busca iterativo encontra o caminho mais curto para a solução aproveitando, ao máximo, o tempo disponibilizado para a busca pela melhor ação [45].

A desvantagem do procedimento iterativo é que ele processa muitos estados repetidos antes de alcançar a profundidade do estado objetivo. Com uma análise mais detalhada, no entanto, percebe-se que, combinando o aprofundamento iterativo com uma tabela de transposição, o custo adicional de processar estados repetidos não afeta a busca em árvores (crescimento exponencial) de maneira significativa, visto que os nós visitados em uma iteração anterior poderão ser buscados em uma tabela de transposição ao invés de serem explorados novamente, conforme será visto com mais detalhe no capítulo 4.

2.5.3 Algoritmo de Busca Alfa-Beta, Aprofundamento Iterativo e Tabela de Transposição

A grande desvantagem do aprofundamento iterativo, como relatado na seção anterior, é o processamento repetido de estados em níveis mais rasos da árvore de busca que acontece antes de se encontrar a profundidade do estado objetivo do problema.

Tabelas de transposição são repositórios, em memória, de estados que foram previamente submetidos ao procedimento de busca [47]. Com o uso de tabelas de transposição, o procedimento de busca em profundidade pode ser modificado de modo a verificar se um

determinado estado encontra-se armazenado em memória antes de explorá-lo (os detalhes poderão ser observados, posteriormente, na seção 4.6.2). Desse modo, a grande desvantagem do aprofundamento iterativo (processamento repetido de estados em níveis mais rasos da árvore de busca) praticamente desaparece, uma vez que buscar informações em memória é um procedimento extremamente rápido.

Além disso, tabelas de transposição são utilizadas em conjunto com aprofundamento iterativo para produzir árvores de buscas parcialmente ordenadas. Entre as informações armazenadas em uma tabela de transposição para um determinado estado, está a melhor ação a ser executada a partir desse estado. Quando o aprofundamento iterativo pesquisar um nível mais profundo da árvore de busca e revisitar um dado estado S_0, o filho S_m de S_0, originado pela execução da melhor ação eventualmente indicada na tabela de transposição para ser executada a partir de S_0, será pesquisado primeiro (ordenação parcial da árvore). Assumindo que uma busca mais rasa é uma boa aproximação para outra mais profunda, a melhor ação para um estado S_0 na profundidade d será, possivelmente, a melhor ação para o estado S_0 na profundidade $d + 1$ [56].

Com o uso de tabelas de transposição e aprofundamento iterativo, o procedimento de busca alfa-beta é aprimorado para expandir árvores em busca pela melhor escolha, tornando tal algoritmo extremamente eficiente e de uso adequado para o ambiente de jogos.

Os detalhes do eficiente algoritmo de busca alfa-beta, tabela de transposição e aprofundamento iterativo serão apresentados no capítulo 4 desta dissertação. Nesse capítulo serão apresentados detalhadamente cada um dos algoritmos que compõe esse método de busca, assim como os problemas encontrados para implementá-lo.

Capítulo 3

Estado da Arte

Pesquisas realizadas pela comunidade científica utilizam o ambiente de jogos para aprimorar e desenvolver técnicas eficientes de inteligência artificial.

Sabendo da vasta aplicabilidade de técnicas inteligentes em aprendizagem em jogos, o objetivo deste capítulo é apresentar algumas das técnicas mais relevantes usadas no domínio de jogos, mais especificamente, no domínio de jogos de tabuleiros. As técnicas abordadas neste capítulo já foram descritas no capítulo 2 em que se trata o referencial teórico que envolve este trabalho.

A criação de programas jogadores com alto nível de desempenho tem sido um dos maiores triunfos da Inteligência Artificial. A exemplo disso, têm-se os sucessos obtidos com os jogos Gamão, Xadrez, Damas e Othello. Esses jogos usam variadas técnicas inteligentes no aprendizado de seus jogadores, e todos com bom nível de competição. Entre os métodos utilizados para obter esses resultados, a busca e a utilização da capacidade de memória dos computadores tem sido uma das técnicas mais bem sucedidas. Nesse sentido, a Aprendizagem por Reforço e a Computação Evolutiva aparecem como opções de técnicas inteligentes para a aquisição autônoma de conhecimento em estratégias de aprendizagem em jogos.

Serão apresentados, neste capítulo, alguns dos melhores projetos científicos na área de jogos de tabuleiro que foram construídos utilizando técnicas inteligentes. Antes, apresentar-se-á uma seção sobre complexidade temporal e espacial de jogos de tabuleiros, a qual tem um objetivo informativo. Após ler esta seção, o leitor terá noção da complexidade em que está inserido o domínio de jogos de tabuleiro, o quanto é custoso, computacionalmente, desenvolver técnicas que trabalhem bem em um ambiente de buscas tão grande.

3.1 Complexidade de Jogos

Em [82] Herik apresenta uma análise exaustiva das principais características dos jogos que mais influenciam em sua complexidade. Em particular, são definidas duas medidas de

complexidade em jogos: *a complexidade do espaço de estados* e a *complexidade da árvore do jogo*.

A complexidade do espaço de estados é definida como o número de posições de jogo legais que podem ser atingidas a partir da posição inicial do jogo. A complexidade da árvore do jogo é definida como o número de folhas na árvore de busca da solução do jogo a partir de uma posição (ou estado) atual. Em outras palavras, a complexidade da árvore do jogo á determinada pelo fator de ramificação do jogo em questão. A principal análise feita em [82] é a de que uma baixa complexidade do espaço de estados é mais importante do que uma baixa complexidade na árvore do jogo como fator determinante para se resolver os problemas dos jogos.

A tabela 3.1, retirada de [13], compara o fator de ramificação e o espaço de estados de alguns jogos de tabuleiro.

Jogo	Ramificação	Estados
Xadrez	30 - 40	10^{50}
Damas	8 - 10	10^{17}
Gamão	± 420	10^{20}
Othello	± 5	$< 10^{30}$
GO 19 x 19	± 360	10^{160}
Abalone	± 80	$< 3^{61}$

Tabela 3.1: Complexidade do espaço de estados e fator de ramificação de alguns jogos de tabuleiro

3.2 Jogadores Automáticos de Damas que utilizam Redes Neurais com Aprendizado por Reforço e Método das Diferenças Temporais

Para que um jogador automático de Damas tenha a eficácia de um perito, é necessário que haja habilidade no tratamento de memória, reconhecimento de padrões e capacidades sofisticadas de planejamento. A maioria dos algoritmos utilizados em jogos de Damas normalmente utilizam uma função de avaliação que retorna uma unidade escalar vinculada a uma dada posição do jogo. Nesse tipo de jogo complexo, devem ser utilizadas técnicas de busca rigorosas em que milhões de posições têm de ser avaliadas antes de se encontrar uma solução viável.

Um dos principais requisitos para se construir um jogador automático de Damas com alto nível de jogo é possuir um bom modelo de função de avaliação. Dado um estado de tabuleiro, a função de avaliação deve ajudar o jogador automático a executar uma ação que traga resultados positivos para si próprio. Quanto maior a quantidade de estados

distintos possíveis em um jogo, mais complexa se torna a tarefa de construir um bom modelo de função de avaliação.

Representar, por meio de funções de avaliação, todas as descontinuidades produzidas pelos diversos estados de tabuleiro disponíveis em um jogo é quase impossível. Então, a utilização de redes neurais tornou-se bastante popular como recurso de aproximação de funções de avaliação [25]. Como exemplo, podem-se citar o Chinook de Schaeffer [71], o *NeuroDraughts* de Lynch [42], [43], o *LS-Draughts* de Neto e Julia [50], [52] e o *VisionDraughts* de Caixeta e Julia [11]. Todos eles utilizam redes neurais como funções de avaliação para treinarem seus jogadores automáticos de Damas.

Para que um agente jogador aprenda a jogar, deve-se aplicar uma estratégia de treinamento à sua função de avaliação de modo que ela possa otimizar as ações do agente e orientá-lo a como se comportar no ambiente de jogo. Entre os vários tipos de treinamentos encontrados hoje na literatura [60], [47], tem-se o treinamento por reforço que ajusta os pesos de uma rede neural de modo a aprimorar o conhecimento dessa rede. Este tipo de treinamento é o mais utilizado no domínio de jogos de tabuleiro em geral.

Segundo SUTTON e BARTO em [77], a Aprendizagem por Reforço significa aprender a jogar de forma a poder, incrementalmente, testar e refinar a função de avaliação. O agente não recebe qualquer informação direta acerca do valor absoluto ou relativo dos exemplos de treino, em vez disso, recebe um sinal escalar do ambiente que lhe indica a eficiência das jogadas efetuadas [50]. O primeiro a construir um sistema de aprendizagem por reforço para o jogo de Damas foi Samuel [61], [62]. No jogador de Samuel, uso-se um algoritmo complexo para selecionar ajustamentos nos parâmetros baseando-se na diferença entre as sucessivas avaliações de posições bem sucedidas em um jogo, a fim de aprender a jogar Damas.

A Aprendizagem por Diferenças Temporais TD (λ) [76], [80] é um caso especial de Aprendizagem por Reforço que fornece um método eficiente para receber exemplos de treino com uma precisão mais elevada, uma vez que a avaliação de um dado estado é ajustada utilizando as diferenças entre a sua avaliação e as avaliações de posições sucessivas. Dessa forma, a previsão do resultado do jogo a partir de uma certa posição está relacionada com as previsões das posições seguintes [50].

Entre os jogadores *Chinook* [68], *NeuroDraughts* [42], *LS-Draughts* [50] e *VisionDraughts* [11], podem-se encontrar jogadores que usam redes neurais com Aprendizagem por Reforço e TD(λ) como auxiliadores no seu processo de aprendizagem e, além disso, contam com interferência humana durante esse processo. Existem também àqueles jogadores que aprendem por si só. As próximas seções tratam, em separado, cada um desses jogadores.

3.2.1 Jogadores com Forte Interferência Humana no Processo de Aprendizagem

Apesar de o processo de aprendizagem da maioria dos jogadores de Damas ser completamente automático, existem jogadores que possuem interferência humana no ajuste das suas funções de avaliação. Como exemplo de um jogador que possui esse tipo de interferência, tem-se o jogador automático de Damas, *Chinook*. Esse jogador, possui uma base de dados com informações perfeitas sobre final de jogo e outra com informações sobre boas jogadas para início de jogo, além disso, possui também ajuste manual em suas funções de avaliação. As bases de dados junto com a função de avaliação com ajuste manual, são as grandes responsáveis por esse jogador se tornar imbatível.

Chinook: Base de Dados e o Ajuste Manual

O projeto *Chinook* foi iniciado em 1989 como uma iniciativa para tentar melhor entender as buscas heurísticas. O *Chinook* se tornou o mais famoso e mais forte jogador de Damas do mundo [63], [71], [68], [67]. Ele é o campeão mundial homem-máquina para o jogo de Damas e usa uma função de avaliação ajustada manualmente para estimar quanto um determinado estado do tabuleiro é favorável para o jogador. Além disso, ele tem acesso à coleções de jogadas utilizadas por grandes mestres na fase inicial do jogo (*opening books*) e um conjunto de bases de dados para as fases finais do jogo com 39 trilhões de estados do tabuleiro (todos os estados com 10 peças ou menos) com valor teórico provado (vitória, derrota ou empate). Para escolher a melhor ação a ser executada, o *Chinook* utiliza um procedimento de busca minimax com poda alfa-beta, aprofundamento iterativo e tabelas de transposição.

O *Chinook* conseguiu o título de campeão mundial de Damas em Agosto de 1994 ao empatar 6 jogos com o Dr. Marion Tinsley que, até então, defendia seu título mundial a mais de 40 anos. Para o *Chinook*, o jogo á dividido em 4 fases e cada uma delas possui 21 características que são ajustadas manualmente para totalizar os 84 parâmetros de sua função de avaliação. Os parâmetros foram ajustados ao longo de 5 anos, por meio de testes extensivos em jogos contra si mesmo e em jogos contra os melhores jogadores humanos.

Diante de tamanho esforço para ajustar a função de avaliação do *Chinook*, Schaeffer [71] e sua equipe realizaram um estudo detalhado de comparação entre uma função de avaliação treinada, manualmente, por peritos e uma função ajustada por Diferenças Temporais [68]. A primeira abordagem do estudo consistiu em treinar os pesos jogando contra o próprio *Chinook* para determinar a eficácia da aprendizagem em face o benefício de jogar contra um oponente de alto desempenho. O segundo conjunto de experiências envolveu o jogo contra si próprio (estratégia de treino por *self-play*) a fim de verificar se a aprendizagem poderia alcançar um alto nível de desempenho sem ter o privilégio de treinar jogando contra um oponente forte (em ambos os casos, o treino foi realizado

utilizando um *look-ahead* de 5, 9 e até 13 níveis de profundidade).

Os resultados do treino por *self-play*, obtidos por Schaeffer, evidenciaram que não é necessário um bom professor para que o programa ajuste um conjunto de pesos de uma função de avaliação de forma a alcançar o nível de jogo de um campeão mundial [68].

Em Damas, é fundamental evitar algumas posições no início de um jogo. Por isso, torna-se necessário a utilização de bases de dados de início de jogos (*openning books*). A falta de um *openning book* em Damas pode fazer com que, antes do quarto movimento, seja encontrada uma situação de derrota forçada. São armadilhas que necessitam de buscas profundas para serem descobertas e, por outro lado, especialistas humanos sabem como enfrentá-las. No início de seu desenvolvimento, o *Chinook* não possuía bases de dados iniciais e, obviamente, não poderia competir no topo, por muito tempo. Tal base de dados demorou a ser inserida no jogador, inclusive, esse foi o motivo de muitas derrotas sofridas no início de sua existência. A necessidade de escolher jogadas corretas motivou a criação do *anti-book* (uma base de dados com movimentos que devem ser evitados no começo do jogo). Essa base de dados foi chamada posteriormente de *openning book* e proporcionou ao jogador maiores condições para executar boas jogadas no início do jogo.

A utilização de bases de dados de finais de jogos em um jogo de Damas tem papel fundamental para a construção de agentes jogadores de alto desempenho. As bases de dados de 8 peças tiveram papel fundamental para o sucesso do *Chinook* contra os melhores jogadores humanos. Elas começaram a ser construídas em 1989 e a construção somente se completou em 1993. Foram mais de 400 bilhões de estados possíveis do tabuleiro comprimidos em mais de 5 gigabytes de dados [40], [69].

As bases de dados do *Chinook* foram construídas de maneira sistemática, tentando provar as posições de vitória ou derrota em um movimento, dois movimentos, três movimentos e assim por diante. Quando acabaram os estados de vitória ou derrota, os estados restantes foram declarados empates. Essas bases de dados não contam com a informação de quantos movimentos são necessários para que o jogador alcance uma vitória ou derrota durante um jogo [70], [40].

Assumindo a eficiência do jogador *Chinook*, as bases de dados de 9 e 10 peças foram iniciadas em 2001 com o intuito de resolver o jogo de Damas. Considerando todas as combinações com 10 peças ou menos sobre o tabuleiro, existem mais de 39 trilhões de estados distintos. Diante de números tão expressivos, o código do *Chinook* teve de ser transformado para utilizar índices de 64 bits de endereçamento. Todos os estados com 10 peças ou menos no tabuleiro tiveram seus valores teóricos (vitória, derrota ou empate) calculados e foram comprimidos em 237 gigabytes. As bases de dados são indispensáveis para a solução do jogo de Damas, e um eficiente algoritmo de compressão é utilizado para permitir rápida localização e descompressão em tempo real [65].

Em 2007, a equipe do *Chinook* anunciou que o jogo de Damas estava resolvido. A partir da posição inicial do jogo, existe uma prova computacional de que o jogo é um empate.

A prova consiste em uma estratégia explícita com a qual o programa nunca perde, isto é, o programa pode alcançar o empate contra qualquer oponente jogando tanto com peças pretas quanto brancas [67].

3.2.2 Jogadores sem Interferência Humana no Processo de Aprendizagem

Nesta seção, serão apresentados os jogadores automáticos de Damas com o aprendizado baseado em Redes Neurais com Aprendizagem por Reforço TD (λ), que não possuem interferência humana no seu processo de aprendizagem. Entre esses jogadores tem-se o *NeuroDraughts* [42], [43], precursor do estudo de jogadores automáticos de Damas pela equipe de Julia [50], [11], o *LS-Draughts* e o *VisionDraughts*, sendo os dois últimos uma evolução do *NeuroDraughts*.

NeuroDraughts e a Busca Minimax

O sistema *NeuroDraughts* de Lynch [42], [43] implementa o agente jogador de Damas como uma rede neural MLP que utiliza a busca minimax para a escolha da melhor jogada em função do estado do tabuleiro do jogo. Além disso, ele utiliza o método de aprendizagem por reforço TD(λ) aliado à estratégia de treino por *self-play* com clonagem, como ferramentas para atualizar os pesos da rede. Para tanto, o tabuleiro é representado por um conjunto de funções que descrevem as características do próprio jogo de Damas. A utilização de um conjunto de características para representar o mapeamento do tabuleiro de Damas na entrada da rede neural é definida por Lynch como sendo um mapeamento *NETFEATUREMAP*.

Em um mapeamento do tipo *NETFEATUREMAP*, o tabuleiro é representado por um determinado número de funções que descrevem as características do próprio jogo de Damas. Cada característica tem um valor absoluto que é convertido em uma sequência binária. Com essa representação, a entrada na rede neural varia de acordo com o número de características utilizadas e a quantidade de dígitos binários que cada característica utiliza para representar seu valor absoluto.

No jogador de Lynch, é a busca minimax que avalia, em conjunto com a rede neural (ou função de avaliação), todos os possíveis movimentos legais detectados para uma determinada posição do jogo corrente. Após essa avaliação, o algoritmo seleciona a ação ou movimento que provê maior predição de vitória. A busca minimax é um método de seleção da melhor ação a ser feita em um jogo, em que dois jogadores se empenham em alcançar objetivos mutuamente exclusivos (*MIN e MAX*), como apresentado no capítulo 2. Ela se aplica especialmente na busca em árvores de jogo para determinar qual a melhor jogada para o jogador atual. O algoritmo se baseia no princípio de que, em cada jogada, esse jogador irá escolher o melhor movimento possível.

Um dos principais objetivos de Lynch quando desenvolveu o *NeuroDraughts* foi, justamente, tentar obter um bom jogador de Damas que pudesse jogar com um alto nível de desempenho sem ter que realizar buscas profundas nem analisar jogos de especialistas humanos [42]

LS-Draughts e o Algoritmo Genético

Dentro do paradigma de aprendizagem por reforço, vários jogadores automáticos utilizam conjuntos de características específicas dos respectivos domínios, como parte fundamental do processo de ajuste de suas funções de avaliação. Em Damas, particularmente, Samuel foi o pioneiro e utilizou 28 características especificas do domínio de jogo de Damas [61]. [62]. Lynch [42], [43] escolheu manualmente 12 dessas características, entre aquelas utilizadas por Samuel, para implementar o *NeuroDraughts*. Tais características podem ser visualizadas na tabela 4.1.

Pensando em automatizar o processo de escolha de características, Neto e Julia desenvolveram o *LS-Draughts*: um sistema que gera, automaticamente, por meio da técnica de algoritmos genéticos, um conjunto de características mínimas necessárias e essenciais para o jogo de Damas, de forma a otimizar o treino de um jogador automático [50]. [51], [52].

Os Algoritmos Genéticos (AGs) foram introduzidos por John Holland na década de 1960 [26], na Universidade de Michigan, com o objetivo de estudar formalmente os conceitos de adaptação que ocorrem na natureza, formalizá-los matematicamente, e desenvolver sistemas artificiais que mimetizam os mecanismos originais encontrados em sistemas naturais.

O *LS-Draughts* aproxima uma rede neural multicamadas por meio do método TD(λ), aliado com busca minimax e a técnica *self-play* com clonagem. O mapeamento do tabuleiro na camada de entrada da rede neural é feito pelo mapeamento NET-FEATUREMAP utilizando o conjunto de características escolhidas pelos algoritmos genéticos. Com o processo de escolha de características automatizado, o *LS-Draughts* supera o nível de jogo alcançado pelo *NeuroDraughts*, conforme pode ser visto nos resultados experimentais mostrados nos trabalhos de Neto e Julia [50]. [51], [52]. A escolha desses conjuntos foi fundamental para se obter bons jogadores durante a etapa de treinamento das redes MLPs, e também para melhorar a eficiência do jogador em relação ao seu predecessor.

O AG possibilitou ao *LS-Draughts* escolher um conjunto mínimo de atributos que melhor caracterizem o domínio de Damas e que sirvam como um meio pelo qual a função de avaliação adquirirá novos conhecimentos, o que é uma questão fundamental para acelerar a aprendizagem e obter novos agentes com alto nível de desempenho.

Os resultados obtidos pelo *LS-Draughts* mostram que é possível jogar Damas com alto nível de desempenho sem ter que utilizar grande quantidade de características. Dessa forma, o AG aparece como uma poderosa ferramenta que auxilia na busca pela melhor combinação de características que possam otimizar o treino por Diferenças Temporais de

um agente jogador de Damas que utiliza este conjunto de características para aprender a jogar.

VisionDraughts e o Eficiente Método de Busca

Enquanto o *NeuroDraughts* e o *LS-Draughts* contam com um sistema básico de busca minimax que utiliza profundidade fixa de busca, o *VisionDraughts* conta com um módulo alfa-beta com tabelas de transposição e aprofundamento iterativo que lhe permite ajustar os pesos de sua rede neural de maneira muito mais precisa, e consequentemente ter mais condições de escolher a melhor jogada em função do estado corrente do jogo. Além disso, ele utiliza o método de Aprendizagem por Reforço TD(λ) aliado à estratégia de treino por *self-play* com clonagem como ferramentas para atualizar os pesos da rede.

Como já citado anteriormente (seção 2.5.1) o algoritmo minimax examina mais estados do tabuleiro que o necessário e, nesse sentido, o mecanismo de combinação do algoritmo minimax com a poda alfa-beta elimina seções da árvore de busca que, definitivamente, não podem conter a melhor ação a ser executada pelo agente jogador. Com esse tipo de mecanismo (poda alfa-beta) o *VisionDraughts* conseguiu uma performance de jogo superior ao obtido pelos jogadores *NeuroDraughts* e *LS-Draughts*.

Ainda no sentido de melhorar o mecanismo de busca do jogador, foi inserido no mesmo o procedimento de aprofundamento iterativo combinado com tabela de transposição. Muitos programas jogadores utilizam aprofundamento iterativo [63], [68], [45], [55]. O uso do aprofundamento iterativo baseia-se na hipótese de que uma busca mais rasa, em jogos de tabuleiro, é uma boa aproximação para uma busca mais profunda.

O procedimento começa pesquisando com profundidade igual a um e termina quase imediatamente. Depois, a profundidade de busca é incrementada passo a passo e o procedimento de busca é realizado para cada um deles. Em virtude ao crescimento exponencial da árvore, o tempo necessário para expandir seu nível mais profundo é muito superior ao tempo necessário para cada um dos níveis mais rasos. Os benefícios mais evidentes do uso do aprofundamento iterativo para os programas jogadores são: obtenção de um mecanismo eficiente de controle de tempo e obtenção de árvores de buscas parcialmente ordenadas (PLAAT et al., 1996b).

As tabelas de transposição são utilizadas em conjunto com o aprofundamento iterativo para alcançar árvores de buscas parcialmente ordenadas. O valor da predição de cada estado já visitado pelo procedimento de busca e o melhor movimento a ser executado a partir dele são armazenados em uma tabela de transposição. Quando o aprofundamento iterativo pesquisar um nível mais profundo da árvore e revisitar estados, o movimento sugerido pela tabela de transposição (caso disponível) será pesquisado primeiro. Assumindo que uma busca mais rasa é uma boa aproximação para outra mais profunda, o melhor movimento da profundidade d será, possivelmente, o melhor movimento para a profundidade $d + 1$ [56].

Outro jogador que utiliza busca em profundidade com aprofundamento iterativo é o *Chinook* [64], [67], porém esse jogador faz uso de um algoritmo de busca paralelo muito eficiente.

3.3 Anaconda e a Computação Evolutiva

A aplicação da Computação Evolutiva em jogos tem se mostrado bastante eficiente na obtenção de bons agentes jogadores, tornando-se, assim, um paradigma alternativo ao processo de treinamento convencional. O processo de treinamento de uma rede neural por meio de Computação Evolutiva proporciona a obtenção de bons jogadores automáticos de Damas. Nesse tipo de treinamento, as melhores soluções tendem a evoluir com o passar das gerações e as piores soluções tendem a desaparecer. Nesse sentido, Fogel [15], [14], [19] utilizou um processo co-evolutivo para implementar um bom jogador de Damas que aprende a jogar sem utilizar perícia humana na forma de características específicas do domínio do jogo.

O melhor jogador de Fogel, chamado Anaconda, foi resultado da evolução de 30 redes neurais multicamadas ao longo de 840 gerações. Cada geração tinha em torno de 150 jogos de treinamento (5 jogos para cada um dos 30 indivíduos da população). Assim, foram necessários 126.000 jogos de treinamento e 6 meses de execução para obtê-lo. Em um torneio de 10 jogos contra uma versão baixo nível do *Chinook*, o Anaconda obteve 2 vitórias, 4 empates e 4 derrotas, o que permitiu classificá-lo como "expert".

Atualmente, escolher um método coevolutivo ou de Aprendizagem por Diferenças Temporais para treinar uma rede neural pode ser uma tarefa árdua. Graças ao sucesso de ambos em alguns domínios específicos como Gamão [16] e Xadrez [20], a conclusão é que ainda existe muito a ser explorado nessa área do conhecimento humano.

Por outro lado, Paul Darwen demonstra em [16] a vantagem de se utilizar Diferenças Temporais no treinamento de redes neurais multicamadas por causa da rapidez com que a rede aprende um comportamento não linear sobre um determinado problema. Darwen demonstra esta questão ao discutir o porquê da coevolução conseguir bater, para uma arquitetura de rede linear (*perceptron* simples), a aprendizagem por Diferença Temporal no jogo do Gamão, mas não conseguir o mesmo feito para uma arquitetura de rede não linear (rede neural com camada oculta). O autor mostra que, se são necessários bilhões de jogos para que uma arquitetura não-linear treinada por um método coevolutivo consiga bater uma outra arquitetura não-linear treinada pelo método TD(λ), a qual, por sua vez, requer apenas cerca de 100.000 jogos para aprender, então muitos dos bilhões de jogos do método coevolutivo não estarão, de fato, contribuindo para a aprendizagem.

Esse fato demonstrado por Paul Darwen parece também ser aplicado ao domínio de Damas, quando se pretende treinar uma rede neural multicamada por meio de um algoritmo coevolutivo. Por exemplo, o melhor jogador de David Fogel (Anaconda) resultou da

evolução de 30 redes neurais multicamadas ao longo de 840 gerações, o que levou 6 meses de execução. Foram necessários 126.000 jogos de treinamento para que Fogel obtivesse o seu melhor jogador, Anaconda. Já em [42], [43], Mark Lynch obteve o seu melhor jogador de Damas depois de apenas 2.000 jogos de treinamento com o método das Diferenças Temporais e utilizando um conjunto de características selecionadas manualmente para representar o mapeamento do tabuleiro de Damas na entrada da rede neural.

3.4 Outros Jogos que Usam Aprendizagem por Reforço e Diferenças Temporais

A aplicabilidade da técnica de Aprendizagem por Reforço e TD (λ) em outros jogos deve-se ao fato de que nesse domínio, o agente de aprendizagem pode ser imaginado como um agente contentor de um elemento de desempenho que decide quais ações executar e um elemento de aprendizagem modificante do elemento de desempenho para que o agente escolha melhores jogadas. Como exemplo de outros bons jogadores que utilizam dessa técnica podem ser citados o Gamão e Xadrez, os quais serão brevemente apresentados a seguir.

3.4.1 O sucesso do *TD-Gammon* para o jogo de Gamão

Gamão é um jogo praticado em um tabuleiro com 24 casas, 15 peças para o jogador preto e 15 peças para o jogador vermelho (todas as peças são do mesmo tipo). Os jogadores se movimentam arremessando 2 dados e seguindo o fluxo mostrado na figura 3.1. O jogo de Gamão é considerado complexo por causa de seu espaço de estados e seu fator de ramificação mostrados na linha 3 da tabela 3.1.

Uma pequena revolução no campo da Aprendizagem por Reforço ocorreu quando Tesauro apresentou os primeiros resultados com o seu jogador de Gamão. O *TD-Gammon*, utilizando pouco conhecimento específico sobre o jogo de Gamão, conseguiu atingir resul-

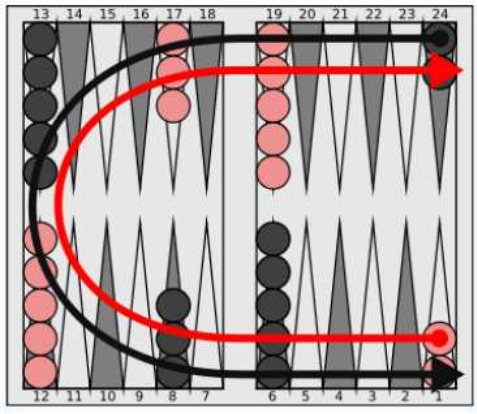

Figura 3.1: Exemplo de um tabuleiro do jogo de Gamão

tados em nível dos maiores jogadores mundiais [78]. [79], [80]. O algoritmo de aprendizagem utilizado no *TD-Gammon* é uma combinação do algoritmo TD(λ) com uma função de aproximação não-linear baseada em uma rede neural multicamadas. Para mapear o tabuleiro do jogo na entrada da rede neural, Tesauro utilizou um conjunto de características do domínio do jogo de Gamão.

Pollack e Blair [57] fundamentaram em 1998 a hipótese de que o sucesso do *TD-Gammon* não está ligado às técnicas de aprendizagem por Diferença Temporal TD(λ), mas sim a uma predisposição inerente à própria dinâmica do jogo de Gamão, assim como à natureza do próprio processo de treino no qual a tarefa muda, dinamicamente, à medida que a aprendizagem ocorre. Os fatores que Pollack e Blair citam como determinantes para o sucesso do *TD-Gammon* são:

1. **rapidez do jogo:** o jogador aprendia a partir de vários jogos contra si próprio (*self-play*). Em treinamentos que utilizam Aprendizagem por Reforço, o número de jogos de treino é importante para o sucesso da aprendizagem;
2. **suavidade na representação:** a avaliação de um estado do tabuleiro no Gamão é uma função razoavelmente suave de posição, facilitando uma boa aproximação por rede neural;
3. **natureza Estocástica:** pelo fato de o Gamão ser jogado com lançamento de dados, isto implica exploração de uma boa parte da quantidade do espaço de estados, forçando o sistema a entrar em regiões desse espaço que ainda não foram vistas pela função de avaliação corrente.

Contudo, e apesar do sucesso do *TD-Gammon* sobre os seus predecessores que foram treinados por aprendizagem supervisionada ou treino por comparação, alguns pesquisadores não concordam que a aprendizagem por Diferença Temporal seja a melhor solução para todos os jogos.

Por outro lado, Sutton demonstrou a convergência do algoritmo TD(λ) para sistemas nos quais a probabilidade de evolução para um determinado estado S_{t+1} só depende do estado atual S_t e da ação A_t selecionada a partir de S_t [76].

3.4.2 Xadrez

Uma abordagem interessante para o jogo de Xadrez foi dada por Baxter e outros pesquisadores [7], [6] a fim de treinar seu jogador de Xadrez *KNIGHTCAP*. O método de treino utilizado foi o TD-*Leaf*(λ), uma variante do algoritmo TD(λ), que permite ao TD-*Leaf* ser usado conjuntamente com a busca minimax para atualizar a função de avaliação do jogador. Esse algoritmo simplesmente usa a posição que surge na folha (daí o seu nome) da árvore de busca minimax para atribuir a predição do estado raiz (estado atual do jogo) e, assim, poder atualizar a função de avaliação do jogador por meio da Diferença

Temporal entre esse estado do jogo atual e o próximo estado. Com essa técnica e jogando contra humanos e computadores pela internet, *KNIGHTCAP* subiu sua classificação ELO (sistema de classificação pontual da Federação Internacional de Xadrez) de 1650 para 2100 em apenas 308 jogos, durante 3 dias [7], [6]. Os ingredientes que contribuíram crucialmente para o sucesso do *KNIGHTCAP* foram a disponibilidade de parceiros de treino em grande variedade no servidor de Xadrez e a integração correta da aprendizagem por TD(λ) nos procedimentos de busca do programa. Em [68], Schaeffer também utiliza esta combinação do algoritmo TD(λ) com a busca minimax, proposto por Baxter, para treinar seu agente jogador de Damas.

Outro jogador de Xadrez que utiliza de Aprendizagem por Reforço e Métodos das Diferenças Temporais no seu processo de aprendizagem é o *NeuroChess* [81]. O *NeuroChess* é um programa que aprende a jogar Xadrez a partir do resultado final dos jogos. O mecanismo de aprendizagem desse jogador é uma rede neural baseada em explanação [48], [49]. Esse jogador conseguiu obter sucesso em vários jogos de Xadrez envolvendo outros jogadores automáticos, porém, quando se trata de jogos envolvendo especialistas humanos, ele é considerado um jogador "pobre", ou seja, fraco.

Como o foco deste trabalho não são outros jogos senão o de Damas, esta seção apenas apresentou um pouco sobre outros jogos que também usam de TD(λ) no seu processo de aprendizagem. Maiores detalhes sobre outros jogos que utilizam dessa técnica podem ser encontrados em: Xadrez [48], [49], [81], Othello [41], Gamão [78], [79], [80] e GO [73].

3.5 Redes Neurais Auto-Organizáveis - *Kohonen-SOM*

Conforme dito anteriormente, as Redes *Kohonen-SOM* possuem aprendizado não-supervisionado, visto que não é necessária a apresentação de uma saída desejada para realizar a correção dos pesos da rede. O processo de aprendizado da rede é o competitivo, isso significa que as unidades competem entre si para ganhar o direito de se ativar [37], [36], [38].

Atualmente, as Redes *Kohonen-SOM* são um dos modelos de rede neural mais importante no paradigma de redes neurais competitivas. São inúmeras as aplicações atuais de uma rede *Kohonen-SOM*. Há, disponível na literatura, uma imensa quantidade de artigos e relatórios publicados nas mais variadas áreas (de engenharia a medicina, de biologia a economia etc). O livro de Kohonen [38] traz várias referências sobre as aplicações desse tipo de rede. No domínio de jogos, no entanto, é difícil encontrar aplicações que fazem uso de redes *Kohonen-SOM*.

Hartmann em [24] desenvolveu o programa *brKChess* o qual auxilia um professor de Xadrez a ensinar tal arte. Esse programa traz à comunidade enxadrística a possibilidade de ensinar tal habilidade por meio de tabuleiros-exemplo. Por meio de um tabuleiro, o professor pode inserir posições de peças diversas e ativar o módulo de cálculo de modo

que ele faça uma pesquisa por tabuleiros com características semelhantes do ponto de vista visual. O programa permite também que alguns parâmetros sejam adaptados pelo professor, a fim de se obterem resultados mais amplos. O sistema *brKChess* utiliza a rede *Kohonen-SOM* para classificar tabuleiros de Xadrez com o intuito de recuperar tabuleiros harmônicos.

Uma das maiores aplicações das redes *Kohonen-SOM* tem sido em reconhecimento de fonemas e de voz. Kohonen [36], [37] apresentou a máquina fonética, *Phonetic Typewriter*, a qual se baseia em uma rede *Kohonen-SOM* bidimensional treinada com partes do espectro de sinais de voz. O sistema é capaz de reconhecimento de fala usando trajetórias de ativações entre os neurônios que são ativados pelos diferentes fonemas.

Várias outras aplicações das redes *Kohonen-SOM* têm sido relatadas nas áreas de mineração de dados e descoberta de conhecimento em bases de dados. A motivação básica é a necessidade de analisar, de forma não-supervisionada, grandes volumes de registros em bancos de dados. O uso de redes *Kohonen-SOM* como ferramenta de mineração de dados tem sido abordada tanto em pesquisa como em produtos, por exemplo, o WEBSOM [33].

O WEBSOM foi desenvolvido objetivando a organização automática de grandes bases de dados de textos, principalmente, os disponíveis na internet, como os disponíveis em listas de grupos de interesse. Esse aplicativo efetua agrupamento nos dados a partir de visualização, ou seja, há necessidade de intervenção do usuário que guia manualmente a escolha dos parâmetros e a segmentação da rede. Nesse projeto, as redes *Kohonen-SOM* foram utilizadas apenas como um instrumento de visualização para indicar tendências de agrupamentos, efetuando a partição manualmente.

Outra área de aplicação das redes *Kohonen-SOM* é a de classificação de padrões de dados ecológicos [53]. Nesse trabalho, Oliveira utiliza a rede de Kohonen como uma metodologia para a classificação das espécies de peixes em categorias tróficas. A rede *Kohonen-SOM* mostrou-se uma ferramenta robusta para a classificação dos dados, apresentando resultados rápidos com uma clara visualização dos agrupamentos, facilitando sobremaneira a interpretação dos resultados. Outros projetos nessa mesma área já haviam sido realizados com o uso do algoritmo de *clusterização K-means* [1], porém os resultados obtidos com as redes *Kohonen-SOM* foram mais favoráveis.

Capítulo 4

VisionDraughts - Um Sistema de Aprendizado de Damas com Eficiente Método de Busca

Neste capítulo, apresenta-se o *VisionDraughts* [12], [11], um sistema de aprendizagem de jogos de Damas cujo principal objetivo foi construir um agente automático capaz de aprender a jogar com alto nível de desempenho e sem auxílio de especialistas humanos. Esse jogador é uma otimização do jogador de Mark Lynch [42], [43] (tratado na sessão 3.2.2), e substitui o algoritmo de busca minimax usado pelo *NeuroDraughts* por um eficiente módulo de busca e uma base de dados com informações sobre final de jogo.

O sistema implementa o agente jogador de Damas como uma rede neural MLP que utiliza a busca com poda alfa-beta, tabela de transposição e aprofundamento iterativo para, em função do estado corrente do jogo, escolher a melhor jogada . Além disso, ele utiliza o método de aprendizagem por reforço TD(λ) aliado à estratégia de treino por *self-play* com clonagem como ferramenta para atualizar os pesos da rede. Para tanto, o tabuleiro é representado por um conjunto de funções que descrevem as características do próprio jogo de Damas. A utilização de um conjunto de características para representar o mapeamento do tabuleiro de Damas na entrada da rede neural é a mesma definida por Lynch em seu jogador *NeuroDraughts* [42] como sendo um mapeamento NET-FEATUREMAP (introduzido na seção 3.2.2 e detalhado na seção 4.1.2).

Dessa forma, para conseguir alto nível de desempenho, o *VisionDraughts* introduziu dois novos módulos em relação ao jogador *NeuroDraughts* [42] e [43], que são: um módulo de busca eficiente em árvores de jogos que fornece ao agente jogador de Damas maior capacidade de analisar jogadas futuras; e um módulo de acesso às bases de dados de finais de jogos [70] que fornece ao agente jogador a capacidade de anunciar, antes do final da partida, se um estado do tabuleiro com até 8 peças representa vitória, derrota ou empate.

As próximas seções abordam, em detalhes, o desenvolvimento do *VisionDraughts*, cuja

compreensão é fundamental para o entendimento do jogador *MP-Draughts* (apresentado no capítulo 5). As seções suceder-se-ão de acordo com o disposto a seguir: i) a representação do tabuleiro nos jogadores de Damas; ii) a arquitetura geral do *VisionDraughts*; iii) o cálculo da predição e a escolha da melhor ação; iv) reajuste de peso da rede MLP; v) e finalmente o eficiente módulo de busca desse jogador.

4.1 Representação do Tabuleiro nos Jogadores de Damas

Existem diversas maneiras de representar internamente um tabuleiro de jogo de Damas [11]. A representação interna de um tabuleiro no jogador *VisionDraughts* faz-se de duas maneiras: representação vetorial e representação por características, também chamada de NET-FEATUREMAP. Essas representações também são usadas no jogador desenvolvido nesse trabalho, o *MP-Draughts*, que será apresentado no capítulo 5.

A representação vetorial é usada para representar o tabuleiro na entrada do algoritmo alfa-beta durante o processo de busca pelo melhor movimento e também na representação de estados de tabuleiros em base de dados. A representação NET-FEATUREMAP é usada na entrada das redes neurais no momento de calcular a predição de um dado estado.

As próximas seções se dedicam a explicar, em detalhes, como é feito cada um dos tipos de representação de tabuleiros citados acima.

4.1.1 Representação Vetorial

A escolha de uma estrutura de dados para representar o tabuleiro em um jogo é fundamental para a eficiência de um jogador automático de Damas. Algoritmos de busca em profundidade, baseados no alfa-beta, estão presentes nos melhores projetos da história dos jogos de tabuleiro; e a escolha adequada de uma estrutura de dados para representar o tabuleiro afeta, consideravelmente, a velocidade de execução desse tipo de algoritmo. A representação vetorial é recomendada para casos em que um algoritmo alfa-beta é usado em virtude de sua precisão na representação de um estado de tabuleiro.

O vetor que representa o estado de tabuleiro de um jogo tem 32 posições correspondentes às posições existentes em um tabuleiro de Damas . O modo de como se faz a representação vetorial de um tabuleiro de Damas é apresentado a seguir:

O tabuleiro do jogo de Damas representado, de maneira simples e direta, por meio de representação vetorial:

```
BOARDVALUES{
  EMPTY = 0,
  BLACKMAN = 1,
```

```
  REDMAN = 2,
  BLACKKING = 3,
  REDKING = 4
}

BOARD{
  BOARDVALUES p[32]
}
```

O tabuleiro é uma estrutura do tipo BOARD implementada como um vetor de 32 elementos do tipo BOARDVALUES. Cada elemento do vetor BOARD representa uma casa do tabuleiro e cada casa do tabuleiro possui um dos valores presentes em BOARDVALUES (EMPTY, BLACKMAN, REDMAN, BLACKKING, REDKING).

4.1.2 Representação NET-FEATUREMAP

A utilização de um conjunto de características para treinar um jogador de Damas foi primeiramente proposta por Samuel [61] com o intuito de prover medidas numéricas para melhor representar as diversas propriedades de posições de peças sobre um tabuleiro. Várias dessas características implementadas por Samuel resultaram de análises feitas sobre o comportamento de especialistas humanos em partidas de Damas. Em termos práticos, essas análises tinham o objetivo de tentar descobrir quais características referentes a um estado do tabuleiro são frequentemente analisadas e selecionadas pelos próprios especialistas quando vão escolher seus movimentos de peças durante uma partida de Damas.

Samuel implementou 28 características referentes ao domínio de jogo de Damas. Caixeta e Julia utilizaram 12 dessas características para representar o tabuleiro do jogo no *VisionDraughts* [11].

Tais características fornecem medidas quantitativas e qualitativas para melhor representar as diversas propriedades das posições das peças sobre um tabuleiro de Damas. Essas características, implementadas por Samuel, podem ser observadas na tabela 4.1 apresentada logo a seguir.

Conjunto de Características implementadas por Samuel

CARACTERÍSTICAS	DESCRIÇÃO FUNCIONAL
PieceAdvantage	Contagem de peças em vantagem para o jogador preto (no caso, número de peças que o jogador tem a mais).
PieceDisadvantage	Contagem de peças em desvantagem para o jogador preto (no caso, número de peças que o jogador tem a menos).
PieceThreat	Total de peças pretas que estão sob ameaça.
PieceTake	Total de peças vermelhas que estão sob ameaça de peças pretas.
Advancement	Total de peças pretas que estão na $5^{\underline{a}}$ e $6^{\underline{a}}$ linha do tabuleiro menos as peças que estão na $3^{\underline{a}}$ e $4^{\underline{a}}$ linha.
DoubleDiagonal	Total de peças pretas que estão na diagonal dupla do tabuleiro.
BackRowBridge	Se existe peças pretas nos quadrados 1 e 3 e se não existem rainhas vermelhas no tabuleiro.
CentreControl	Total de peças pretas no centro do tabuleiro.
XCentreControl	Total de quadrados no centro do tabuleiro onde tem peças vermelhas ou para onde elas possam mover.
TotalMobility - MOB	Total de quadrados vazios para onde as peças vermelhas podem mover.
Exposure	Total de peças pretas que são rodeadas por quadrados vazios em diagonal.
Taken	Total de posições para qual uma peça vermelha pode se mover e, assim, poder ameaçar uma peça preta em um movimento subsequente.
Pole	Total de peças simples pretas que estão completamente rodeada de posições vazias.
Node	Total de peças vermelhas que estão rodeada por, no mínimo, 3 posições vazias.

KingCentreControl	Total de rainhas pretas no centro do tabuleiro.
TriangleofOreo	Se não existe rainhas vermelhas e se o triângulo de Óreo (posições 2,3 e 7) está ocupado por peças pretas, ou se não existe rainhas pretas e se o triângulo de Óreo (posições 26,30 e 31) esta ocupado por peças vermelhas.
Threat	Total de posições para qual uma peça preta pode se mover e assim poder ameaçar uma peça vermelha em um movimento subsequente.
UndeniedMobility	Esta característica é a diferença entre *MOB* e *DenialofOccupancy*.
Hole	Total de posições vazias que estão rodeadas por 3 ou mais peças.
BackRowControl	Se não existem rainhas pretas e, se, uma ponte ou um triângulo de Oreo é ocupado por uma peça vermelha
Gap	Total de posições vazias separadas por duas peças vermelhas ao longo de uma diagonal ou que separam uma peça vermelha de uma margem do tabuleiro.
Exchange	Total de posições que a peça preta pode avançar uma casa e forçar o oponente a comer uma peça.
DiagonalMoment	Total de peças vermelhas que estão localizadas na posição 1 ou na posição 2 de uma diagonal dupla mais as peças passivas que estão na ponta da diagonal.
Apex	Se não existe rainha preta no tabuleiro, ou se os quadrados 7 ou 26 estão ocupados por uma peça preta e se nenhum desses quadrados estão ocupado por uma peça vermelha.
Cramp	Se ocorre uma limitação de posição para as peças vermelhas e pelo menos uma das posições estão ocupadas por peças pretas.
Dyke	Total de peças vermelhas que ocupem três posições de diagonal adjacentes.

DenialofOccupancy	Total de posições onde, no próximo movimento, uma peça preta pode ocupar esta posição e ser capturada na próxima jogada.
Fork	Total de peças vermelhas que ocupam duas posições adjacentes em uma coluna e que nessa mesma coluna, existam três posições vazias; assim a peça preta, pela ocupação de uma dessas posições pudo capturar uma ou duas peças vermelhas.

Tabela 4.1: Conjunto de 28 Características implementadas por Samuel

A representação NET-FEATUREMAP é mais conveniente do que a representação vetorial em alguns casos como, por exemplo, na representação do tabuleiro de jogo de Damas na entrada das redes neurais no momento de calcular a predição de um dado estado, uma vez que fornece, também, uma visão qualitativa do tabuleiro que é mais apropriada ao processo de sua avaliação do que a visão meramente posicional e quantitativa da representação vetorial.

Sendo assim, uma coleção de n características fornece uma representação quantitativa e qualitativa de um conjunto B de estados de tabuleiros. Cada característica f_i representa um certo atributo de um determinado tabuleiro x e pode ser definida como:

$f_i : B \longrightarrow \mathbb{N}$, onde:

$f_i(\mathrm{x}) = a$, em que a representa o atributo do tabuleiro x referente à característica f_i, conforme exemplificado abaixo.

Considere f_i como sendo a característica peças em vantagem "*PieceAdvantage*", que verifica em um dado tabuleiro x a quantidade de peças em vantagem de um jogador. Se o jogador para o qual a característica está sendo verificada possui 5 peças em vantagem em relação ao seu oponente, o atributo $f_i(x)$ produz 5 (ou seja, a =5); caso ele não possua peças em vantagem, o valor do atributo $f_i(x)$ é 0 (ou seja, a =0). Se um tabuleiro possui peças em vantagem para um jogador, consequentemente, ele não possui peças em desvantagem para esse mesmo jogador. A figura 4.1 ilustra como seria a representação do tabuleiro em relação às características *PieceAdvantage* e *PieceDisadvantage*, ambas ocupando a 1ª e a 2ª posição do tabuleiro, respectivamente, considerando 5 peças em vantagem.

Conforme visto anteriormente, a conveniência de se usar representação vetorial ou representação NET-FEATUREMAP depende do processo em foco. Para as situações em que a representação NET-FEATUREMAP é mais adequada, faz-se necessário um processo de conversão de representação vetorial para tal representação (módulo NET-

Figura 4.1: Exemplo de tabuleiro representado por NET-FEATUREMAP

FEATUREMAP apresentado na arquitetura do jogador), que é definido pelo mapeamento C abaixo:

C: B $\longrightarrow \mathbb{N}^n$, onde:

C(x) = $< f_1$(x),..., f_n(x) $>$

Em que n é a quantidade de características usadas na representação do tabuleiro por características, a serem escolhidas entre as 28 características propostas por Samuel [61], [62]. Isso significa que cada estado de tabuleiro x é representado por uma *n-tupla* composta de n atributos, os quais correspondem as características f_1,..., f_n, respectivamente.

4.2 Arquitetura Geral do *VisionDraughts*

O fluxo mostrado na figura 4.2 ilustra a arquitetura do jogador *VisionDraughts*. Conforme já comentado, esse jogador consiste de uma rede neural MLP treinada pelo método de aprendizagem por reforço TD(λ). Esse método utiliza predições (ou estimativas de resultados finais do jogo) sucessivas para ensinar a rede neural a jogar Damas. O jogador alcança alto nível de desempenho simplesmente jogando contra si mesmo (estratégia de treino por *self-play* com clonagem) com uma busca eficiente e com miníma análise de jogos de especialistas humanos.

Os módulos no qual a arquitetura, figura 4.2, se divide são:

1. **Rede Neural Multicamadas**: a rede neural recebe, em sua camada de entrada, um estado do tabuleiro do jogo e devolve, em seu único neurônio na camada de saída, um valor real compreendido entre -1.0 e +1.0. Tal valor (predição) representa o quão o estado do tabuleiro presente na camada de entrada da rede é favorável ao jogador automático;
2. **Módulo *NET-FEATUREMAP***: por meio do mapeamento C, esse módulo converte a representação vetorial do estado corrente em representação NET-FEATUREMAP e apresenta esta última à de entrada da rede neural MLP. Como o *VisionDraughts* usa 12 características para representar o tabuleiro, tem-se que C: B $\rightarrow \mathbb{N}^{12}$. Cada característica tem um valor absoluto que representa a sua medida analítica sobre um determinado estado do tabuleiro. Tal valor absoluto é convertido (conversão numérica entre as bases decimal e binária) em *bits* significativos que, em conjunto com os demais *bits* das outras características presentes na conversão, constituem

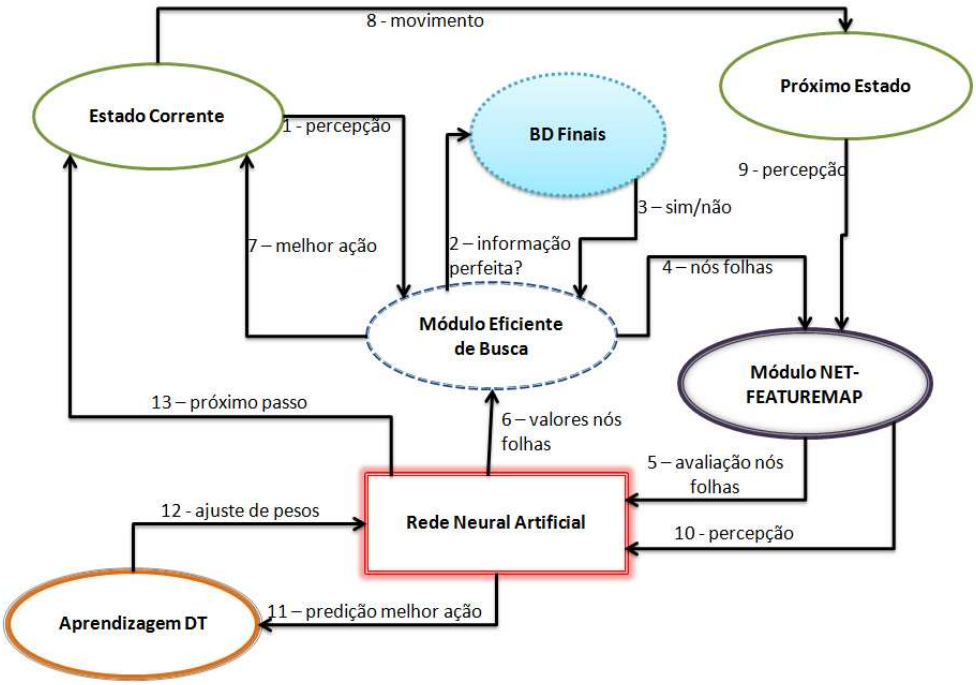

Figura 4.2: Fluxo de aprendizado do *VisionDraughts*: um sistema de aprendizagem de jogos de Damas

a sequência binária de saída do módulo NET-FEATUREMAP a ser apresentada na entrada da rede neural. As características implementadas no *VisionDraughts*, juntamente com as quantidades de bits significativos de cada uma, podem ser vistos na tabela 4.2:

CARACTERÍSTICAS	N° BITS
PieceAdvantage	4
PieceDisadvantage	4
PieceThreat	3
PieceTake	3
Advancement	3
DoubleDiagonal	4
BackRowBridge	1
CentreControl	3
XCentreControl	3
TotalMobility - MOB	4
Exposure	3
KingCentreControl	3

Tabela 4.2: Conjunto de Características implementadas no jogador *VisionDraughts*

3. **Módulo Eficiente de Busca**: seleciona a melhor ação a ser executada pelo agente em função do estado corrente do tabuleiro. Esse módulo expande uma árvore em busca da melhor ação e os nós presentes na camada mais profunda (folhas) são

mapeados, pelo módulo 2 na entrada da rede neural do módulo 1;

4. **Módulo Ajuste de Peso**: o ajuste dos pesos da rede neural é feito pelo método das Diferenças Temporais TD(λ). Esse processo é responsável pela aquisição de conhecimento do sistema;
5. **Base de Dados**: O uso de bases de dados de finais de jogos [70], [40] reduz o número de movimentos necessários a partir da jogada inicial, para se alcançar posições com valor teórico definido (vitória, derrota ou empate). O *VisionDraughts* tem acesso às bases de dados disponibilizadas em [69] e à biblioteca com funções de acesso disponibilizada em [23].

Quase todos os programas para jogos de tabuleiro possuem a mesma arquitetura básica [54]. Suas diferentes partes precisam gerar os próximos movimentos legais, avaliá-los e retornar o melhor movimento a ser executado pelo agente jogador.

Observe que, para o problema de treinar uma rede neural com a finalidade de jogar Damas utilizando o método TD(λ) e mapeamento *NET-FEATUREMAP* do tabuleiro, a escolha da melhor ação está vinculada a dois fatores fundamentais: a profundidade de busca (que indica a capacidade que o agente jogador possui de analisar combinações futuras de jogadas) e a função de avaliação (que avalia uma determinada posição do tabuleiro do jogo). Como observado por Neto e Julia [50], [51], [52], a ocorrência do problema do *loop* de final de jogo indica certa imprecisão da função de avaliação. Assim sendo, o *VisionDraughts* utiliza bases de dados de finais de jogos para reduzir a ocorrência do problema do *loop* e aprimorar a função de avaliação.

4.3 Cálculo da Predição e Escolha da Melhor Ação

O cálculo das predições é efetuado por uma rede neural artificial acíclica com 3 camadas. O número de nodos na camada de entrada da rede, varia de acordo com o número de características e *bits* significativos presentes no módulo de mapeamento. A camada oculta é formada por 20 neurônios fixos e a camada de saída por um único neurônio. A arquitetura geral da rede neural é mostrada na figura 4.3. Nessa arquitetura, cada um dos neurônios da rede está conectado a todos os outros das camadas subsequentes (amplamente conectada); um termo *bias* é utilizado para todos os neurônios da camada oculta e outro para o neurônio da camada de saída.

Formalmente, o processo de cálculo da predição P_t referente a uma configuração do tabuleiro do jogo de Damas em um instante temporal t, isto é, S_t, pode ser descrito da seguinte forma:

1. a representação interna do tabuleiro S_t é mapeada na entrada da rede neural pelo módulo de mapeamento mostrado na figura 4.2;

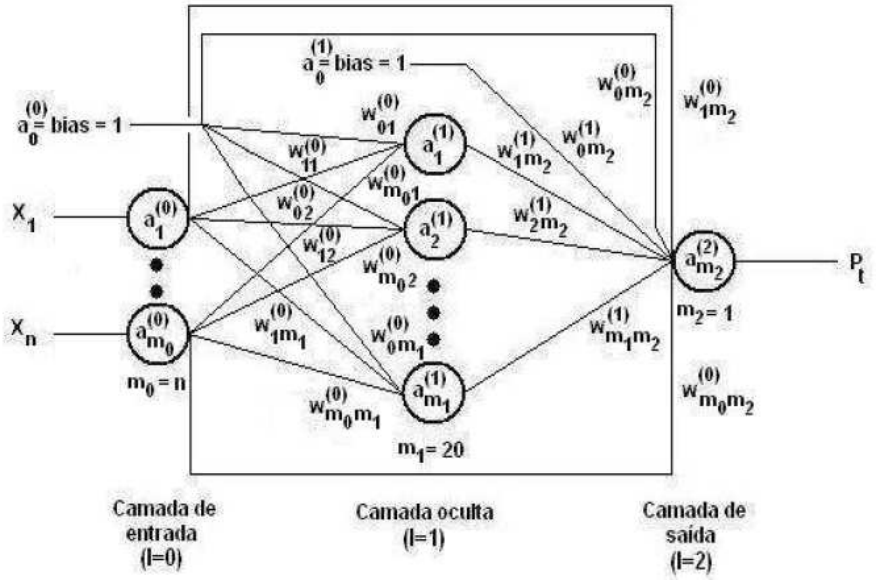

Figura 4.3: Rede Neural utilizada pelo *VisionDraughts*

2. calcula-se o campo local induzido $in_j^{(l)}$ para o neurônio j (valor de entrada para o neurônio j) na camada l, para $1 \leq \mathrm{l} \leq 2$, da seguinte forma:

$$in_j^l = \begin{cases} \sum_{i=0}^{m_{(l-1)}} w_{ij}^{(l-1)}.a_i^{(l-1)}, & \text{para neurônio } j \text{ na camada } l{=}1 \\ \sum_{i=0}^{m_{(l-1)}} w_{ij}^{(l-1)}.a_i^{(l-1)} + \sum_{i=0}^{m_{(l-2)}} w_{ij}^{(l-2)}.a_i^{(l-2)}, & \text{para neurônio } j \text{ na camada } l{=}2 \end{cases}$$

   onde m_l representa neurônios na camada l; a_i^l é o sinal de saída do neurônio i na camada l e w_{ij}^l é o peso sináptico da conexão de um neurônio i da camada l com o neurônio j das camadas posteriores à camada l. Para as camadas ocultas ($l = 1$) e de saída ($l = 2$) sendo $i = 0$, tem-se que $a_0^{(l-1)} = +1$ e $w_{0j}^{(l-1)}$ é o peso do bias aplicado ao neurônio j na camada l;

3. obtido o campo local induzido, o sinal de saída do neurônio j, na camada l, para $1 \leq \mathrm{l} \leq 2$, é dado por $a_j^l = g_j(in_j^{(l)})$, onde $g_j(x)$ é uma função de ativação tagente hiperbólica definida por $g_j(x) = \frac{2}{(1+e^{-2x})} - 1$;

4. a predição retornada pela rede neural para o estado S_t é $a_j^{(2)} = a_{m2}^{(2)} = P_t$.

Funcionalmente, predições P_t's calculadas pela rede neural MLP podem ser vistas como uma estimativa do quão o estado S_t se aproxima de uma vitória (representada pelo retorno do valor +1 pelo ambiente), derrota (representada pelo retorno do valor -1 pelo ambiente) ou empate (representado pelo retorno do valor 0, ou próximo de 0, pelo ambiente). Assim,

configurações de tabuleiros (ou estados do jogo) que receberem predições próximas de +1 tenderão a ser consideradas como bons estados de tabuleiro, resultantes de boas ações, que poderão convergir para vitória (+1). Da mesma forma, tabuleiros cujas predições estão próximas de -1 tenderão a ser considerados péssimos estados de tabuleiro, resultantes de ações ruins, que poderão convergir para derrota (-1). O mesmo vale para configurações de tabuleiros próximos de 0, que poderão convergir para empate (0 ou valor próximo deste).

O problema de escolher a melhor ação a ser executada pelo jogador *VisionDraughts* é solucionado por meio de uma rede neural artificial, utilizando o cálculo das predições e do algoritmo alfa-beta. O alfa-beta recebe como parâmetro de entrada o estado corrente do tabuleiro do jogo, expande uma árvore de busca e envia suas folhas para o módulo de mapeamento que gera a sequência binária de entrada na rede neural. O pseudo-código do algoritmo alfa-beta será apresentado na seção 4.6.1

4.4 Reajuste de Pesos da Rede Neural MLP

O reajuste dos pesos da rede é *online*, isto é, o agente vai jogando contra o seu clone (oponente) por meio do treinamento por *self-play* com clonagem e os pesos da rede vão sendo ajustados pelo método TD(λ) de acordo com a escolha das melhores ações e os estados resultantes dessas ações. Após o fim de cada partida de treino, um reforço final é fornecido pelo ambiente informando o resultado obtido pelo agente jogador em função da sequência de ações que executou (+1 ou -1, caso o resultado tenha sido vitória ou derrota, respectivamente). Caso tenha ocorrido empate, o resultado será zero ou valor próximo disso.

O agente jogador seleciona a melhor ação M_t a ser executada a partir de um estado S_t com o auxílio do procedimento de busca alfa-beta e dos pesos atuais da rede neural. O estado S_{t+1} resulta da ação M_t sobre o estado S_t. A partir de então, o estado S_{t+1} é mapeado na entrada da rede neural e tem sua predição P_{t+1} calculada (a predição é o valor de saída no neurônio da última camada da rede neural). Os pesos da rede neural são reajustados com base na diferença entre P_{t+1} e a predição P_t, calculada anteriormente para o estado S_t. Após o fim de cada partida de treino, um reforço final é fornecido pelo ambiente informando o resultado obtido pelo agente jogador em função da sequência de ações que executou (+1 para vitória, -1 para derrota e um valor próximo de 0 para empate).

Formalmente, o cálculo do reajuste dos pesos é definido pela equação do método TD(λ)

de Sutton (SUTTON, 1988):

$$\begin{aligned} w_{ij}^{(l)} &= w_{ij}^{(l)}(t-1) + \Delta w_{ij}^{(l)}(t) \\ &= w_{ij}^{(l)}(t-1) + \alpha^{(l)}.(P_{t+1} - P_t).\sum_{k=1}^{t} \lambda^{t-k} \nabla_w P_k \\ &= w_{ij}^{(l)}(t-1) + \alpha^{(l)}.(P_{t+1} - P_t).elig_{ij}^{(l)}(t), \end{aligned} \tag{4.1}$$

onde $\alpha^{(l)}$ é o parâmetro da taxa de aprendizagem na camada. Caixeta e Julia [11] utilizou uma mesma taxa de aprendizagem para todas as conexões sinápticas de uma mesma camada l); $w_{ij}^{(l)}(t)$ representa o peso sináptico da conexão entre a saída do neurônio i da camada l e a entrada do neurônio j da camada $(l+1)$ no instante temporal t. A correção aplicada a esse peso no instante temporal t é representada por $\Delta w_{ij}^{(l)}(t)$; o termo $elig_{ij}^{(l)}(t)$ é único para cada peso sináptico $w_{ij}^{(l)}(t)$ da rede neural e representa o traço de eligibilidade das predições calculadas pela rede para os estados resultantes de ações executadas pelo agente desde o instante temporal 1 do jogo até o instante temporal t; $\nabla_w P_k$ representa a derivada parcial de P_k em relação aos pesos da rede no instante k. Cada predição P_k é uma função dependente do vetor de entrada $\overrightarrow{X(k)}$ e do vetor de pesos $\overrightarrow{W(k)}$ da rede neural no instante temporal k. O termo λ^{t-k}, para $0 \leq \lambda \leq 1$, tem o papel de dar uma "pesagem exponencial"para a taxa de variação das predições calculadas a k passos anteriores de t. Quanto maior for λ, maior o impacto dos reajustes anteriores ao instante temporal t sobre a atualização dos pesos $w_{ij}^{(l)}(t)$.

Neto e Julia [50], [51], [52] descrevem o processo de reajuste de pesos por Diferenças Temporais TD(λ) nas seguintes etapas:

1. o vetor $\overrightarrow{W(k)}$ de pesos é gerado aleatóriamente;
2. as eligibilidades associadas aos pesos da rede são inicialmente nulas;
3. dadas duas predições sucessivas P_t e P_{t+1}, referentes a dois estados consecutivos S_t e S_{t+1}, calculadas em consequência de ações executadas pelo agente durante o jogo, define-se o sinal de erro pela equação:

   $$e(t) = (\gamma P^{t+1} - P^t),$$

   onde o parâmetro γ é uma constante de compensação da predição P_{t+1} em relação a predição P_t;
4. cada eligibilidade $elig_{ij}^{(l)}(t)$ está vinculada a um peso sináptico $w_{ij}^{(l)}(t)$ correspondente. Assim, as eligibilidades vinculadas aos pesos da camada l, para $0 \leq l \leq 1$, no instante temporal t $elig_{ij}^{(l)}(t)$ são calculadas observando as equações dispostas a seguir:

- para os pesos associados às ligações diretas entre as camadas de entrada ($l = 0$) e saída ($l = 2$):

$$elig_{ij}^{(l)}(t) = \lambda.elig_{ij}^{(l)}(t-1) + g'(P_t).a_i^{(l)},$$

onde λ tem o papel de dar uma "pesagem exponencial"para a taxa de variação das predições calculadas a k passos anteriores de t;$a_i^{(l)}$ o sinal de saída do neurônio i na camada l;$g'(x) = (1 - x^2)$ representa a derivada da função de ativação (tangente hiperbólica) [43].

- para os pesos associados às ligações entre as camadas de entrada ($l = 0$) e oculta ($l = 1$):

$$elig_{ij}^{(l)}(t) = \lambda.elig_{ij}^{(l)}(t-1) + g'(P_t).w_{ij}^{(l)}(t).g'(a_j^{(l+1)}).a_i^{(l)},$$

onde $a_j^{(l+1)}$ é o sinal de saída do neurônio j na camada oculta $(l + 1)$;

- para os pesos associados as ligações entre as camadas oculta ($l = 1$) e de saída ($l = 2$):

$$elig_{ij}^{(l)}(t) = \lambda.elig_{ij}^{(l)}(t-1) + g'(P_t).a_i^{(l)};$$

5. calculados as eligibilidades, a correção dos pesos $w_{ij}^{(l)}(t)$ da camada l, para $0 \leq l \leq 1$, é efetuada através da seguinte equação:

$$\Delta w_{ij}^{(l)}(t) = \alpha^{(l)}.e(t).elig_{ij}^{(l)}(t), \tag{4.2}$$

onde o parâmetro de aprendizagem $\alpha^{(l)}$ é definido por Caixeta [11] como:

$$\alpha^{(l)} = \begin{cases} \dfrac{1}{n}, & \text{para } l{=}0 \\ \\ \dfrac{1}{20}, & \text{para } \ l{=}1 \end{cases}$$

6. existe um problema típico associado ao uso de redes neurais no qual a convergência não é garantida para o melhor valor [50]. Caixeta [11], assim como Lynch [42], [43] utilizou o termo momento μ para tentar solucionar esse tipo de problema. Para isso, ele empregou uma checagem de direção na equação 4.1 ou seja, o termo momento μ é aplicado somente quando a correção do peso atual $\Delta w_{ij}^{(l)}(t)$ e a correção anterior $\Delta w_{ij}^{(l)}(t-1)$ estiverem na mesma direção. Portanto, a equação final TD(λ) utilizada para calcular o reajuste dos pesos da rede neural na camada l, para$0 \leq l \prec 1$, é definida por:

$$w_{ij}^{(l)}(t) = w_{ij}^{(l)}(t-1) + \Delta w_{ij}^{(l)}(t); \tag{4.3}$$

onde $\Delta w_{ij}^{(l)}(t)$ é obtido nas seguintes etapas:

- calcule $\Delta w_{ij}^{(l)}(t)$ pela equação 4.1;
- se ($\Delta w_{ij}^{(l)}(t) > 0$ e $\Delta w_{ij}^{(l)}(t-1) > 0$) ou $\Delta w_{ij}^{(l)}(t) < 0$ e $\Delta w_{ij}^{(l)}(t-1) < 0$), então faça:

$$\Delta w_{ij}^{(l)}(t) = \Delta w_{ij}^{(l)}(t) + \mu \Delta w_{ij}^{(l)}(t-1);$$

4.5 Estratégia de Treino por *Self-Play* com Clonagem

A ideia do treinamento por *self-play* com clonagem é treinar um jogador por vários jogos contra ele mesmo, ou seja, uma cópia de si próprio. À medida que o jogador melhora seu nível de desempenho de forma a conseguir bater sua cópia, uma nova clonagem é realizada e o jogador passa a treinar contra esse novo clone. O processo se repete por um determinado número de jogos de treinamento.

Na prática, o treinamento do agente jogador *VisionDraughts* segue os passos:

1. primeiro, os pesos da rede neural MLP do jogador *opp1* (nome atribuído a rede original) são gerados aleatoriamente;
2. antes de iniciar qualquer treinamento, a rede *opp1* é clonada para gerar a rede clone *opp1-clone*;
3. inicia-se então o treinamento da rede *opp1* que joga contra a rede *opp1-clone* (oponente). As redes começam uma sequência de n jogos de treinamento. Somente os pesos da rede *opp1* são ajustados durante os n jogos;
4. ao final dos n jogos de treinamento, um torneio de dois jogos é realizado para verificar qual das redes é a melhor. Caso o nível de desempenho da rede *opp1* supere o nível da rede *opp1-clone*, é realizada a cópia dos pesos da rede *opp1* para a rede *opp1-clone*. Caso contrário, não se copiam os pesos e a rede original *opp1* permanece inalterada para a próxima sessão de treinamento;
5. vá para etapa 3 e execute uma nova sessão de n jogos de treinamento entre a rede *opp1* e o seu último clone *opp1-clone*. Repita o processo até que um número máximo de jogos de treinamento (parâmetro do jogo) seja alcançado.

Observe que essa estratégia de treinamento utilizada no *VisionDraughts* é eficiente, uma vez que o agente jogador *opp1* deve sempre procurar melhorar o seu nível de desempenho a cada sessão de n jogos de treinamento, de forma a poder bater seu clone *opp1-clone* em um torneio de dois jogos. No primeiro jogo do torneio, a rede *opp1* joga com as peças pretas do tabuleiro de Damas e a rede *opp1-clone* joga com as peças vermelhas. Já no segundo jogo, as posições de ambas as redes sobre o tabuleiro de Damas são invertidas. O objetivo dessa troca de posições dos jogadores sobre o tabuleiro é permitir uma melhor avaliação do desempenho das redes *opp1* e *opp1-clone*, ao jogarem entre si

em ambos os lados do tabuleiro, uma vez que as características referem-se a restrições sobre peças pretas e/ou vermelhas.

4.6 Eficiente Método de Busca do *VisionDraughts*

Quando se fala que o *VisionDraughts* [11] é uma otimização do *NeuroDraughts* [42], na verdade, quer-se dizer que o *VisionDraughts* expande o *NeuroDraughts* por incluir dois módulos na arquitetura dele. Esse módulos, como citado anteriormente, são: um eficiente algoritmo de busca baseado em poda alfa-beta, tabela de transposição e aprofundamento iterativo e uma base com informações perfeitas sobre final de jogo.

A qualidade de um programa jogador depende muito do número de jogadas que ele pode olhar adiante (*look-ahead*), o que é aumentado com o uso do módulo de busca eficiente [11]. O jogador automático tem maior capacidade de analisar jogadas futuras(estados do tabuleiro mais distantes do estado corrente) e, consequentemente, realizar melhores jogadas.

Como confirmação do ganho obtido pelo aperfeiçoamento do módulo de busca, tem-se que: com a introdução do algoritmo alfa-beta, o *VisionDraughts* treinou um jogador automático de Damas em apenas 5,83% do tempo gasto pelo *NeuroDraughts*; além disso, usando o alfa-beta combinado com tabelas de transposição e aprofundamento iterativo, esse tempo caiu para 2.27% do tempo total gasto pelo *NeuroDraughts*.

Para verificar o impacto do módulo eficiente de busca no desempenho do *VisionDraughts*, dois torneios com 14 jogos foram executados entre o *VisionDraughts* e os jogadores automáticos de Damas *NeuroDraughts* [42] e *LS-Draughts* [50]. Os resultados dos torneios comprovaram, além da otimização do tempo de busca obtida pelo alfa-beta, a eficiência do aumento do *look-ahead*:

- *VisionDraughts* x *NeuroDraughts*: 5 vitórias para o *VisionDraughts*, 8 empates e 1 derrota;
- *VisionDraughts* x *LS-Draughts*: 4 vitórias para o *VisionDraughts*, 8 empates e 2 derrotas.

Na sequência serão apresentados os "módulos"que compõem o eficiente módulo de busca do jogador. A sequência da apresentação foi organizada de modo a facilitar o entendimento do leitor, visto que a mesma ordem apresentada a seguir, foi seguida no momento da implementação do método. A ordem de apresentação será: i)O algoritmo Alfa-Beta; ii) A Tabela de Transposição; iii) Integração entre o Algoritmo Alfa-Beta e a Tabela de Transposição, iv) O Aprofundamento Iterativo; v) e finalmente O Algoritmo Alfa-Beta com a Tabela de Transposição e o Aprofundamento Iterativo (peças fundamentais para o entendimento do módulo de busca eficiente).

4.6.1 O algoritmo Alfa-Beta

O jogo de Damas pode ser pensado como uma árvore de possíveis estados futuros do tabuleiro do jogo, ou seja, uma árvore representando uma evolução dos próximos movimentos disponíveis de acordo com as regras do jogo. Geralmente a raiz da árvore tem vários filhos representando todas as possíveis jogadas permitidas a partir do estado corrente(raiz da árvore). A profundidade máxima de busca nessa árvore é determinada baseando-se na quantidade de recursos computacionais disponíveis. Os estados do tabuleiro presentes na camada mais profunda da árvore de busca denominam-se folhas da árvore (para simplificar a representação da árvore de busca, cada estado do tabuleiro é representado, de agora em diante, por um círculo chamado nó da árvore de busca).

O *VisionDraughts* utiliza o algoritmo alfa-beta para escolher, na árvore de busca, a melhor ação a ser executada de acordo com o estado corrente do tabuleiro do jogo. O algoritmo alfa-beta pode ser resumido como um procedimento recursivo que escolhe a melhor jogada a ser executada fazendo uma busca em profundidade, da esquerda para a direita, na árvore do jogo [72], [8]. A versão mais conhecida do algoritmo alfa-beta é a versão *hard-soft*, apresentada nesta seção.

Com o objetivo de simplificar o entendimento do algoritmo alfa-beta, apresenta-se, a seguir, o pseudo-código do algoritmo minimax juntamente com a explicação de como ele funciona. Em seguida, será adicionado o mecanismo de poda alfa-beta no contexto do procedimento minimax. Na verdade, o que acontece, de fato, é a introdução de uma poda alfa-beta(limites que guiam as podas) no minimax; por isso o entendimento do minimax faz-se necessário.

Pseudo-código do algoritmo minimax

```
1: minimax(node:n, int:depth, move:bestmove)
2: if leaf(n) or depth=0 then
3:    return evaluate(n)
4: end if
5: if n is a max node then
6:    besteval := - infinity
7:    for each child of n: do
8:       v := minimax (child,d-1,bestmove)
9:       if v > besteval then
10:         besteval:= v
11:         thebest = bestmove
12:      end if
13:   end for
14:   bestmove = thebest
15:   return besteval
```

```
16: end if
17: if n is a min node then
18:    besteval := + infinity
19:    for each child of n: do
20:       v := minimax (child,d-1,bestmove)
21:       if v < besteval then
22:          besteval:= v
23:          thebest = bestmove
24:       end if
25:    end for
26:    bestmove = thebest
27:    return besteval
28: end if
```

- **linha 1:** o algoritmo recebe como parâmetro de entrada um estado do tabuleiro do jogo de Damas n, uma profundidade de busca d e um parâmetro de saída *bestmove* que armazena a melhor jogada a ser executada a partir do estado do tabuleiro n. Ele retorna a predição associada ao estado n na forma de um número real, a qual é a avaliação do estado n do ponto de vista do agente jogador;
- **linhas de 2 a 4**: o algoritmo minimax é recursivo e escolhe a melhor jogada a ser executada fazendo uma busca em profundidade, da esquerda para a direita, na árvore do jogo. Por se tratar de um procedimento recursivo, exige-se uma condição de parada. Tal condição de parada do algoritmo verifica se o estado do tabuleiro n é uma folha da árvore, ou seja, se n não possui filhos. Nos dois casos, a função *evaluate(n)* é retornada indicando a predição dada pela rede neural para o estado do tabuleiro n;
- **linha 5**: o estado do tabuleiro presente na raiz da árvore do jogo montada pelo algoritmo é um nó maximizador e a predição associada a um nó maximizador será igual à maior predição de seus filhos. Os filhos do estado do tabuleiro presente na raiz da árvore do jogo são nós minimizadores e a predição associada a um nó minimizador será igual à menor predição de seus filhos. Assim, os níveis da árvore do jogo alternam entre nós maximizadores e minimizadores e o pedaço de código delimitado por esta linha é executado para os nós maximizadores;
- **linha 6:** *besteval* representa a melhor avaliação encontrada para o nó n até o presente momento. Como n representa um nó maximizador, inicialmente, o valor de *besteval* é configurado como o maior valor negativo possível, por exemplo, -*infinity*. Note que *besteval* recebe -*infinity* e será incrementado até o máximo valor das predições associadas aos filhos de n;

- **linha 7:** para cada um dos filhos de *n*, realiza-se os procedimentos das linhas 8 a 12;
- **linha 8:** Para cada um dos filhos *child*, do estado do tabuleiro *n*, o algoritmo minimax é chamado, recursivamente, com profundidade *d - 1*. O valor da predição associada ao filho *child* de *n* é armazenado na variável *v*;
- **linhas de 9 a 12:** caso a predição armazenada na variável *v* seja maior que *besteval* (melhor avaliação encontrada até o momento para *n*), o algoritmo atualiza o valor de *besteval* com *v* e o valor de *thebest* (melhor movimento encontrado até o momento para *n*) com *bestmove*;
- **linha 14:** assim que o algoritmo sair do laço da linha 7, *thebest* conterá o melhor movimento a ser executado a partir do estado *n*. Então, *bestmove* é atualizado com o valor de *thebest*;
- **linha 15:** Assim que o algoritmo sair do laço da linha 7, *besteval* conterá a maior predição de todos os filhos do estado do tabuleiro *n* (*n* é maximizador). Então, *besteval* será retornado como valor da predição associada ao estado do tabuleiro *n*;
- **linha 17:** o pseudo-código delimitado por esta linha é similar ao delimitado pela linha 5, porém, tratam-se, agora, de nós minimizadores.
- **linha 18:** *besteval* representa a melhor avaliação encontrada para o nó *n* até o momento. Como *n* representa um nó minimizador, inicialmente, o valor de *besteval* é configurado como o maior valor positivo possível, por exemplo, *+ infinity*. Note que *besteval* recebe *+ infinity* e será decrementado até o mínimo valor das predições associadas aos filhos de *n*;
- **linha 19:** para cada um dos filhos de *n*, realizam-se os procedimentos das linhas 20 a 24;
- **linha 20:** para cada um dos filhos *child*, do estado do tabuleiro *n*, o algoritmo é chamado, recursivamente, com a profundidade de busca decrementada de uma unidade. O valor da predição associada ao filho *child* de *n* é armazenada na variável *v*.
- **linhas de 21 a 24:** caso a predição armazenada na variável *v* seja menor que *besteval* (melhor avaliação encontrada até o momento para *n*), o algoritmo atualiza o valor de *besteval* com *v* e o valor de *thebest* (melhor movimento encontrado até o momento para *n*) com *bestmove*;
- **linha 26:** assim que o algoritmo sair do laço da linha 19, *thebest* conterá o melhor movimento a ser executado a partir do estado *n*. Então, *bestmove* é atualizado com o valor de *thebest*;

- **linha 27:** assim que o algoritmo sair do laço da linha 19, *besteval* conterá a menor predição de todos os filhos do estado do tabuleiro n (n é minimizador). Então, *besteval* será propagado, ou seja, retornado como valor da predição associada ao estado do tabuleiro n.

Como citado anteriormente, o algoritmo minimax examina mais estados do tabuleiro que o necessário e, nesse sentido, o mecanismo de poda alfa-beta elimina seções da árvore de busca que, definitivamente, não podem conter a melhor ação a ser executada pelo agente jogador. Abaixo, é apresentado o pseudo-código do algoritmo alfa-beta seguido de uma explicação dos pontos em que ele difere do minimax.

Pseudo-código do algoritmo alfa-beta *hard-soft*

```
1: alfaBeta(node:n, int:depth, int:min, int:max, move:bestmove)
2: if leaf(n) or depth=0 then
3:    return evaluate(n)
4: end if
5: if n is a max node then
6:    besteval := min
7:    for each child of n: do
8:       v := alphaBeta(child,d-1,besteval,max,bestmove)
9:       if v > besteval then
10:         besteval:= v
11:         thebest = bestmove
12:      end if
13:   end for
14:   if besteval >= max then
15:      return max
16:   end if
17:   bestmove = thebest
18:   return besteval
19: end if
20: if n is a min node then
21:   besteval := max
22:   for each child of n: do
23:      v := alphaBeta(child,d-1,min,besteval,bestmove)
24:      if v < besteval then
25:         besteval:= v
26:         thebest = bestmove
27:      end if
28:   end for
```

```
29:    if besteval <= min then
30:       return min
31:    end if
32:    bestmove = thebest
33:    return besteval
34: end if
```

- **linha 1:** enquanto o algoritmo minimax recebe um estado do tabuleiro do jogo de Damas *n*, uma profundidade de busca *d* e um parâmetro de saída para armazenar a melhor ação a ser executada, *bestmove* a partir de *n*, o algoritmo alfa-beta recebe, ainda, um intervalo de busca delimitado pelos parâmetros *min* (representando o limite inferior do intervalo de busca, sendo também conhecido como alfa) e *max* (representando o limite superior do intervalo de busca, sendo também conhecido como beta). A predição de retorno do algoritmo alfa-beta para o estado do tabuleiro presente na raiz da árvore de busca é exatamente igual à predição retornada pelo algoritmo minimax;
- **linha 8:** para cada um dos filhos *child*, do estado do tabuleiro *n*, o algoritmo alfa-beta é chamado, recursivamente, com profundidade *d - 1*. O valor da predição associada ao filho child de *n* é armazenado na variável *v*. Note, porém, que um novo intervalo de busca será utilizado para a chamada recursiva: em vez de utilizar o intervalo [*min; max*], será utilizado o intervalo [*besteval; max*]. Tal fato significa que, no nível de maximização, sempre que ocorrer atualização de um dos limites do intervalo de busca, ela será um incremento no limite inferior. Isso acontece sempre que a predição *v* calculada para *child* superar o valor de *besteval* na linha 9;
- **linhas de 14 a 16:** se acontecer de a predição armazenada na variável *besteval* ultrapassar o limite superior do intervalo de busca ($besteval \geq \max$), o algoritmo retornará, imediatamente, o limite superior do intervalo de busca (*max*) como predição associada ao estado do tabuleiro *n*. Tal fato expressa a ideia de que o intervalo de busca deve ser respeitado, ou seja, a predição associada ao estado do tabuleiro *n* deve estar contida dentro do intervalo de busca passado como parâmetro;
- **linha 23:** para cada um dos filhos *child*, do estado do tabuleiro *n*, o algoritmo minimax é chamado, recursivamente, com profundidade *d - 1*. O valor da predição associada ao filho *child* de *n* é armazenado na variável *v*. Note, porém, que um novo intervalo de busca será utilizado para a chamada recursiva: em vez de utilizar o intervalo [*min; max*], será utilizado o intervalo [*min; besteval*]. Tal fato significa que, no nível de minimização, sempre que ocorrer atualização de um dos limites do intervalo de busca, ela será um decremento no limite superior. Isso acontece sempre que a predição *v* calculada para *child* for inferior ao valor de *besteval* na linha 24;

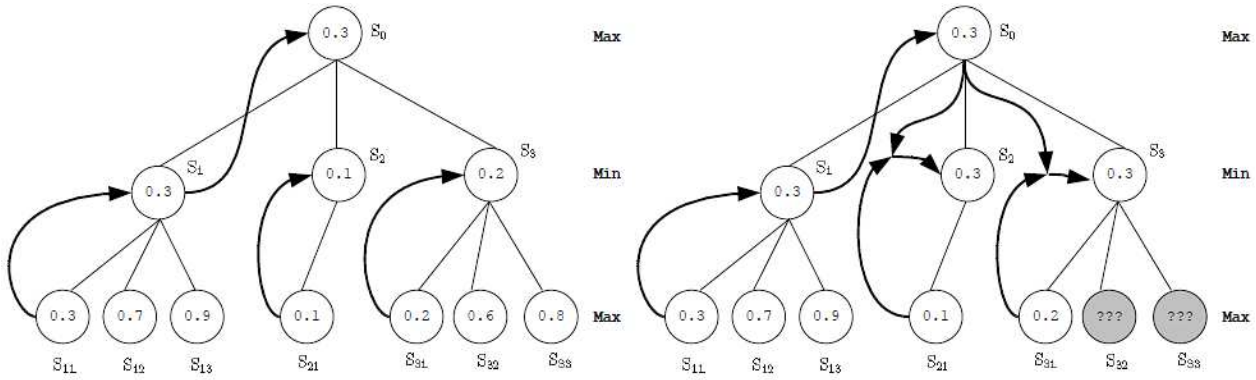

Figura 4.4: A esquerda: árvore de busca expandida pelo minimax. A direita: árvore expandida pelo Alfa-Beta.

- **linhas de 29 a 31:** se acontecer de a predição armazenada na variável *besteval* tornar-se menor que o limite inferior do intervalo de busca ($besteval \leq min$), o algoritmo retornará, imediatamente, o limite inferior do intervalo de busca (min) como predição associada ao estado do tabuleiro n. Tal fato expressa a ideia de que o intervalo de busca deve ser respeitado, ou seja, a predição associada ao estado do tabuleiro n deve estar contida dentro do intervalo de busca passado como parâmetro.

Para melhor esclarecer como funciona a poda alfa-beta durante um processo de busca pela melhor ação (no caso de Damas, melhor movimento), observe na figura 4.4 :

Considerando o exemplo de árvore do jogo de Damas mostrado na figura 4.4, o algoritmo minimax executaria a seguinte sequência de busca pelos nós da árvore , S_1, S_{11}, S_{12}, S_{13}, S_2, S_{21}, S_3, S_{31}, S_{32} e por último S_{33}. Como é possível perceber, o algoritmo percorre todos os nós da árvore e, só ao final, retorna o melhor movimento, no caso S_1. Já se tratando do algoritmo alfa-beta, a sequência de busca seria a seguinte S_1, S_{11}, S_{12}, S_{13}, S_2, S_{21}, S_3, S_{31} e, nesse momento, a busca seria interrompida (porque o valor mínimo encontrado na árvore 0.2 já é menor que o valor mínimo permitido pelo intervalo de busca que é 0.3) e o melhor movimento seria retornado, assim como no minimax,a melhor ação seria se mover para S_1.

Portanto, com base no exemplo mostrado na figura 4.4 e no pseudo-código do algoritmo alfa-beta, é possível abstrair a seguinte ideia para o mecanismo de poda: a avaliação dos filhos de um nó de minimização pode ser interrompida tão logo a predição calculada para um de seus filhos seja menor que o parâmetro alfa (poda alfa). Similarmente, a avaliação dos filhos de um nó de maximização pode ser interrompida tão logo a predição calculada para um de seus filhos seja maior que o parâmetro beta (poda beta).

Como citado anteriormente, o primeiro avanço do *VisionDraughts* em relação ao seu predecessor, *NeuroDraughts*, foi conseguido com o algoritmo alfa-beta, o qual diminuiu o tempo de execução do *VisionDraughts* para um valor inferior á 10% do tempo exigido pelo algoritmo minimax do *NeuroDraughts*.

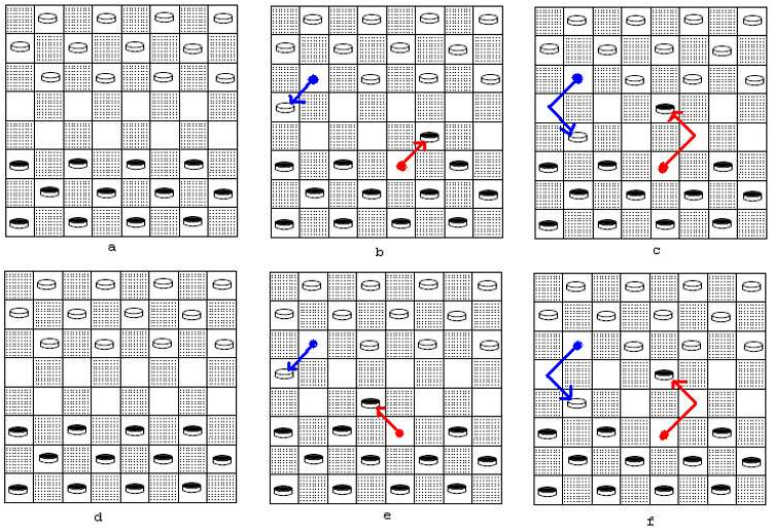

Figura 4.5: Exemplo de transposição em c e f : o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com peças simples.

A fim de proporcionar ainda mais eficiência ao mecanismo de busca, o *VisionDraughts* utiliza o algoritmo alfa-beta em conjunto com tabela de transposição. A subseção seguinte introduz o conceito de transposição e descreve o funcionamento de uma tabela de transposição.

4.6.2 A Tabela de Transposição

O algoritmo alfa-beta apresentado na seção anterior não mantém um histórico dos estados da árvore de jogo já procurados anteriormente. Assim, se um estado do tabuleiro for apresentado 2 vezes para o algoritmo alfa-beta, a mesma rotina será executada 2 vezes a fim de encontrar a predição associada ao estado. Para evitar que o algoritmo alfa-beta seja executado duas vezes para encontrar uma mesma predição associada ao estado, pode-se associá-lo com uma tabela de transposição [10], [9].

Sabe-se que em um jogo de Damas, pode-se chegar a um mesmo estado do tabuleiro 2 vezes (ou mais); quando isso ocorre, diz-se que houve uma transposição [47]. Essas transposições acontecem, em um jogo de Damas, de duas formas básicas:

- diferentes combinações de jogadas com peças simples: as peças simples não se movem para trás, apesar disso, elas podem desencadear uma transposição, como mostrado na figura 4.5. Nesse caso, os estados do tabuleiro mostrados em *a* e *d* são idênticos, assim como os estados mostrados em *c* e *f*. Assumindo *a* como estado inicial, é possível alcançar *c* passando por *b*. Assumindo *d* como estado inicial, é possível alcançar *f* passando por *e*. Então, os únicos estados diferentes são *b* e *e*. No caso da sequência de movimentos *a*, *b* e *c*, o jogador preto move-se primeiro para a direita e, em seguida, para a esquerda, enquanto na sequência de movimentos *d*, *e* e *f*, o jogador preto move-se primeiro para a esquerda e, em seguida, para a direita;

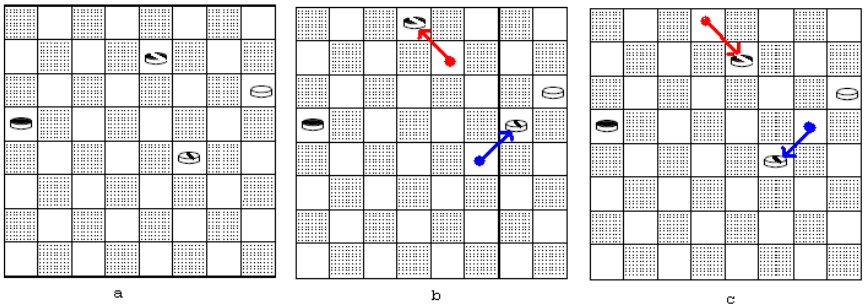

Figura 4.6: Exemplo de transposição em a e c: o mesmo estado do tabuleiro é alcançado por combinações diferentes de jogadas com reis.

- diferentes combinações de jogadas com reis: os reis se movem em qualquer direção, gerando transposições facilmente, conforme mostrado na figura 4.6. Partindo do estado *a*, avançando o rei, é possível alcançar o estado *b* e, em seguida, recuando o rei, é possível alcançar o estado *c*, idêntico ao *a*.

Para entender melhor como o *VisionDraughts* utiliza a tabela de transposição, as subseções, a seguir, abordarão em detalhes como isso é feito. Como a compreensão da técnica de Zobrist é fundamental para a construção da tabela de transposição, inicialmente, apresenta-se a técnica de Zobrist [88]. Na sequência, são apresentados a estrutura da tabela de transposição, e como tratar possíveis colisões de estados do tabuleiro dentro da tabela.

Técnica de Zobrist - Criação de Chaves Hash para Indexação dos Estados do Tabuleiro do Jogo

A tabela de transposição utilizada pelo *VisionDraughts* foi implementada como uma tabela *hash*. Uma tabela *hash* é uma estrutura de dados que associa *chaves* a *valores* [60]. Cada *chave* representa um estado do tabuleiro do jogo de Damas e é associada à informações relevantes obtidas, a partir do algoritmo alfa-beta, para aquele estado. A representação de um determinado estado do tabuleiro do jogo, na forma de uma chave *hash*, é feita utilizando a técnica descrita por Zobrist [88].

O método descrito por Zobrist utiliza o operador **XOR** (ou exclusivo), simbolizado matematicamente por $\otimes$. Logicamente, o **XOR** é um tipo de disjunção lógica entre dois operandos que resulta em "verdadeiro"se, e somente se, exatamente, um dos operandos tiver o valor "verdadeiro". Computacionalmente, o operador **XOR** pode ser aplicado sobre dois operandos numéricos [11]. Por exemplo:

1. operandos numéricos na base binária: o XOR aplicado sobre dois bits quaisquer resulta em "1"se, e somente se, exatamente, um dos operandos tiver o valor "1". Assim, considere $Seq_1 = b_1, b_2, ..., b_n$ uma sequência binária de n bits. Além disso, considere $Seq_2 = r_1, r_2, ..., r_n$ outra sequência binária, também, de n bits. Para

calcular $Seq_3 = Seq_1 \otimes Seq_2$, basta aplicar o operador XOR sobre os bits das posições correspondentes de Seq_1 e Seq_2, isto é, basta fazer $Seq_3 = b_1 \otimes r_1, b_2 \otimes r_2, ..., b_n \otimes rn$;

2. operandos numéricos na base decimal: a operação XOR sobre dois inteiros decimais segue o mesmo procedimento mostrado para operandos numéricos na base binária, exceto, os dois argumentos inteiros decimais que devem ser, antes de tudo, convertidos para a base binária. A conversão de inteiros decimais para binários é transparente em C++ [1]. Isso significa que dois operandos inteiros decimais podem ser passados como argumentos para o operador XOR (a conversão é feita implicitamente).

Assuma as seguintes propriedades, descritas em [88], para o operador XOR aplicado sobre sequências aleatórias (r) de inteiros decimais de n bits:

1. $r_i \otimes (r_j \otimes r_k) = (r_i \otimes r_j) \otimes r_k$;
2. $r_i \otimes r_j = r_j \otimes r_i$;
3. $r_i \otimes r_i = 0$;
4. se $s_i = r_1 \otimes r_2 \otimes ... \otimes r_i$ então s_i é uma sequência aleatória de n bits;
5. s_i é uniformemente distribuída (uma variável é dita uniformemente distribuída quando assume qualquer um dos seus valores possíveis com a mesma probabilidade).

Suponha que exista um conjunto finito S qualquer e que se deseje criar chaves *hash* para os subconjuntos de S. Um método simples seria associar inteiros aleatórios de n bits aos elementos de S e, a partir daí, definir a chave *hash* de um subconjunto S_0 de S como sendo o resultado da operação $\otimes$ sobre os inteiros associados aos elementos de S_0. Pelas propriedades 1 e 2, a chave *hash* é única e, pelas propriedades 4 e 5, a chave *hash* é aleatória e uniformemente distribuída. Se qualquer elemento for adicionado ou retirado do subconjunto S_0, a chave *hash* mudará pelo inteiro que corresponde àquele elemento.

No caso do *VisionDraughts*, existem 2 tipos distintos de peças (peça simples e rei), 2 cores distintas de peças (peça preta e peça branca) e 32 casas no tabuleiro do jogo. Então, existem, no máximo, 128 possibilidades distintas (2x2x32) de colocar alguma peça em alguma casa do tabuleiro. Assim, criou-se um vetor de 128 elementos inteiros aleatórios, mostrado na figura 4.7, para representar os estados possíveis do tabuleiro (cada elemento representa uma possibilidade de se ocupar uma das 32 casas do tabuleiro com alguma das 4 peças inerentes ao jogo). A chave *hash*, para representar cada estado do tabuleiro, é o resultado da operação **XOR** realizada entre todos os elementos do vetor associados às casas não vazias do tabuleiro. Na figura, pode-se visualizar que são criadas 4 chaves para cada posição do tabuleiro, o que cobre a necessidade de variações de chaves em um jogo de Damas.

[1] linguagem usada na implementação do jogador

RANDOM INT64	PIECE	SQUARE	RANDOM INT64	PIECE	SQUARE
14787540466645868636	black man	1	...		...
2120251484556677534	white man		...		
584882445155849028	black king		...		
3760951787791404667	white king		...		
17903615704209920410	black man	2	8978665553187022367	black man	25
5781218707178284009	white man		6792129980026176469	white man	
7894141919871615785	black king		11106003084864057887	black king	
3578131985066232389	white king		5684749757081299935	white king	
1817657397089932766	black man	3	3967728617316940461	black man	26
9537396155164801519	white man		16232032669744814011	white man	
5808583100557493539	black king		13546780321862426801	black king	
3651659200175719294	white king		3009792841844867034	white king	
11250323712845617096	black man	4	13422590923753360614	black man	27
15592542546949822810	white man		10221763887329211198	white man	
16204138130260099375	black king		5616157223557226974	black king	
9585321403807695269	white king		2865046354894257591	white king	
15915542026527195059	black man	5	14642594631129895935	black man	28
16248679709773236148	white man		8381146724961928037	white man	
6685379756495787903	black king		30233076556323211181	black king	
6977407078633077238	white king		8375086150794650026	white king	
1729081295984380347	black man	6	11810041679881260088	black man	29
6892212846999406827	white man		1213308520865758682	white man	
632708781781195948	black king		9734715559513728574	black king	
8082145037705841596	white king		12184937488032720561	white king	
11740811010298599996	black man	7	4993510297519374450	black man	30
3489214435435855631	white man		12124137870041646186	white man	
14579749940077582302	black king		2664161134633443445	black king	
6486449913624012919	white king		327774891080306970	white king	
3466492341137833191	black man	8	14888968537176605210	black man	31
471079928059731524	white man		6271745259985944523	white man	
12658037930106435315	black king		14507257672045050736	black king	
11963310641682407293	white king		8740695389947450601	white king	
...		...	9487810991141940225	black man	32
...			14639527447367762922	white man	
...			8795549574004575914	black king	
...			18030604617695974466	white king	

Figura 4.7: Vetor de 128 elementos inteiros aleatórios utilizados pelo *VisionDraughts*

A técnica de Zobrist é, provavelmente, o método disponível mais rápido para calcular uma chave *hash* associada a um estado do tabuleiro do jogo de Damas [88]. Os fundamentos dessa informação é a velocidade com que se executa a operação *XOR* por uma CPU. Para entender como ocorre a atualização incremental das chaves *hash*, associadas aos estados do tabuleiro, considere as movimentações possíveis no jogo de Damas:

1. movimento simples: um movimento simples pode ser tratado como a remoção de uma peça simples da casa de origem do movimento e sua inserção na casa de destino do movimento;
2. promoção: uma promoção acontece quando uma peça simples se torna rei. Pode ser tratada como sendo a remoção de um tipo peça e a inserção de outro tipo de peça na mesma casa do tabuleiro;
3. captura: uma captura pode ser tratada como sendo a remoção da peça capturada, a remoção da peça capturadora da casa de origem do movimento e a inserção da peça capturadora na casa de destino do movimento.

Considere o exemplo de movimento simples da figura 4.8 e o vetor de números aleatórios

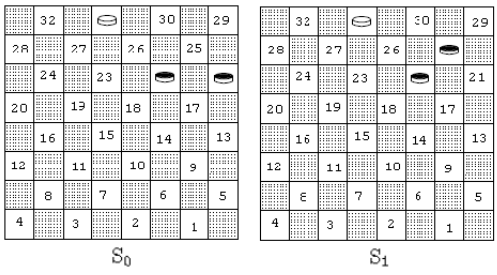

Figura 4.8: Exemplo de movimento simples.

utilizados pelo *VisionDraughts*, mostrado na figura 4.7:

1. para conseguir o número aleatório associado à peça preta simples, localizada na casa 21 do tabuleiro S_0, basta fazer uma consulta ao vetor V e encontrar o valor $I_{21} = 1171196056361380757$;
2. para conseguir o número aleatório associado à peça preta simples, localizada na casa 22 do tabuleiro S_0, basta fazer uma consulta ao vetor V e encontrar o valor $I_{22} = 9204715365712158256$;
3. para conseguir o número aleatório associado à peça branca simples, localizada na casa 31 do tabuleiro S_0, basta fazer uma consulta ao vetor V e encontrar o valor $I_{31} = 6271745259985944523$;
4. para conseguir a chave hash C_0, associada ao estado do tabuleiro S_0, basta considerar $C_0 = I_{21} \otimes I_{22} \otimes I_{31}$. Logo, $C_0 = 1171196056361380757 \otimes 9204715365712158256 \otimes 6271745259985944523$, ou seja, $C_0 = 4104165011015584366$;
5. para conseguir a chave hash C_1, associada ao estado do tabuleiro S_1, não é necessário repetir os passos anteriores; basta atualizar, incrementalmente, o valor da chave C_0 conforme passos 6 e 7;
6. $C_1 = C_0 \otimes I_{21}$. Indica a remoção da peça preta simples da casa 21 do estado S_0 (veja a propriedade número 3). Logo, $C_1 = 4104165011015584366 \otimes 1171196056361380757$, ou seja, $C_1 = 2932970144385327611$;
7. $C_1 = C_1 \otimes I_{25}$. Isso indica a inserção da peça preta simples na casa 25 do tabuleiro, gerando o estado S_1 (veja a propriedade número 1). Logo, $C_1 = 2932970144385327611 \otimes 8978665553187022367$, ou seja, $C_1 = 8988798927364941823$.

Estrutura ENTRY - Dados Armazenados para um Determinado Estado do Tabuleiro do Jogo

A tabela de transposição utilizada pelo *VisionDraughts*, implementada como uma tabela *hash* a fim de que se consiga máxima velocidade de manipulação, mantém um registro de estados analisados do tabuleiro do jogo com respectivos valores obtidos para

os estados pelo algoritmo alfa-beta. A partir de então, sempre que um estado do tabuleiro for apresentado como entrada do procedimento alfa-beta, primeiro o algoritmo verificará se esse estado se encontra na tabela de transposição e, caso afirmativo, utilizará os valores armazenados em memória, o que abreviará todo o procedimento de busca.

O *VisionDraughts* utiliza uma estrutura chamada ENTRY para armazenar os dados de entrada para a tabela de transposição. Veja:

```
struct ENTRY{
  int64   hashvalue;
  int     scorevalue;
  MOVE    bestmove;
  int     depth;
  int     scoretype;
  int     checksum;
}
```

Onde: o campo *hashvalue* armazena a chave *hash* mostrada na seção acima; o campo *bestvalue* armazena o valor da predição retornada pelo algoritmo alfa-beta; o campo *bestmove* armazena a melhor jogada que é, também, retornada pelo algoritmo alfa-beta; o campo *depth* armazena a profundidade de busca utilizada pelo algoritmo alfa-beta; O campo *scoretype* armazena um *flag* que indica o real significado da predição contida em *bestvalue*. Esse flag pode assumir três valores diferentes, *hashAtMost* (quando acontece uma poda alfa), *hashAtLeast* (quando acontece uma poda beta) e *hashExact* quando não acontece nenhuma poda e o valor de *bestvalue* é exato. O campo *checksum* armazena outra chave *hash*, gerada de forma idêntica à chave *hashvalue*, porém, partindo de números aleatórios diferentes. A chave *checksum* utiliza números inteiros de 32 bits enquanto *hashvalue* utiliza inteiros de 64 bits.

A estrutura ENTRY é a unidade básica de entrada para a tabela de transposição utilizada pelo *VisionDraughts*. A tabela de transposição é uma estrutura do tipo *TTABLE* que contém dois vetores (e_1 e e_2, ambos com elementos do tipo *ENTRY*), um método para armazenar novas entradas e um método para ler as entradas já existentes na tabela de transposição. Um exemplo de como é armazenada uma estrutura *ENTRY* na tabela pode ser visto logo abaixo:

```
struct ENTRY{
  int64 hashvalue = hv;
  int bestvalue = 0.3;
  MOVE bestmove = S1;
  int depth = 2;
  int scoretype = hashExact;
  int checksum = ck;
```

```
}
```

A próxima seção se dedica a explicar os possíveis problemas de colisão de registros na tabela de transposição e utiliza esta segunda chave *hash*, *checksum*, para o mesmo estado do tabuleiro, como um dos instrumentos para detecção e tratamento de colisões. A estrutura TTABLE será detalhada na seção 4.6.2, porém, é necessário entender, primeiro, o mecanismo de tratamento de colisões utilizado pelo *VisionDraughts*.

Colisões - Conflitos de Endereços para Estados do Tabuleiro do Jogo

A tabela de transposição do *VisionDraughts* trata dois tipos de erros identificados por Zobrist [88]. O primeiro tipo de erro, chamado erro *tipo 1*, ocorre quando dois estados distintos do tabuleiro do jogo de Damas são mapeados na mesma chave *hash*. O erro *tipo 1* ocorre quando dois estados distintos do tabuleiro são mapeados na mesma chave *hash*. Caso o erro não seja detectado, pode acontecer de predições incorretas serem retornadas pela rotina de busca alfa-beta com tabela de transposição.

Para controlar os erros *tipo 1*, são usadas as duas chaves *hash* da estrutura ENTRY : *hashvalue* e *checksum*. Cada uma das chaves é gerada utilizando números aleatórios independentes. Se dois estados diferentes do tabuleiro produzirem a mesma chave *hash*, *hashvalue*, é improvável que produzam, também, o mesmo valor para *checksum*. A segunda chave *hash* não precisa ser, necessariamente, do mesmo tamanho da primeira. Porém, quanto maior o número de bits presentes nas chaves *hash*, menor a probabilidade de ocorrência dos erros *tipo 1*. A primeira chave *hash*, *hashvalue*, contém 64 bits e a segunda, *checksum*, possui 32 bits.

O erro *tipo 2* ocorre em decorrência dos recursos finitos de memória existentes, quando dois estados distintos do tabuleiro, apesar de serem mapeados em chaves *hash* diferentes, são direcionados para o mesmo endereço na tabela de transposição.Em seguida é apresentada a estratégia utilizada pelo *VisionDraughts* para resolver o dilema dos erros *tipo 2*.

O *VisionDraughts* utiliza dois esquemas de substituição quando acontece uma colisão do *tipo 2*, um chamado *Deep* e outro chamado *New*. Esses esquemas foram escolhidos entre os sete estudados por Breuker em [10], [9]. A escolha desses dois esquemas foi feita baseada no jogador *Chinook* [71], [67]. O *Chinook* utiliza uma tabela de transposição de dois níveis em que cada entrada da tabela pode conter informações relativas a até 2 estados do tabuleiro (um estado em cada nível). Experimentos no Chinook mostram que tal estrutura para a tabela de transposição reduz o tamanho da árvore de busca em até 10% [64].

Caso um endereço da tabela de transposição armazene informações sobre o mesmo estado do tabuleiro S_0 em seus dois níveis, isso indica que as informações sobre S_0 do primeiro nível teriam sido obtidas a partir de uma busca mais profunda que aquela a

partir da qual foram obtidas as informações sobre S_0 armazenadas no segundo nível. Caso um endereço da tabela de transposição armazene, em seus dois níveis, informações sobre dois estados distintos do tabuleiro, isso indica que o segundo nível foi utilizado para resolver um problema de colisão. Em termos práticos, o primeiro nível armazena os dados de maior precisão enquanto o segundo, age como um 'cache"temporal.

Então, a tabela *TTABLE* contém 2 vetores *e1* e *e2*, do tipo *ENTRY*, representando os 2 níveis da tabela de transposição, e cada esquema de substituição está associado a um dos dois vetores de *TTABLE*, como mostrado abaixo:

- ***Deep***: o esquema de substituição *Deep* é tradicional e baseado nas profundidades das sub-árvores examinadas para as posições envolvidas. Em uma colisão, a posição com a mais profunda sub-árvore é preservada na tabela. O conceito por trás desse esquema é que uma sub-árvore mais profunda, normalmente, contém mais nós do que uma sub-árvore mais rasa, e tal fato faz com que o algoritmo alfa-beta economize um tempo maior quando é poupado de pesquisar uma sub-árvore mais profunda;
- ***New***: o esquema de substituição *New* substitui o conteúdo de uma posição na tabela quando uma colisão ocorre. Tal conceito é baseado na observação de que a maioria das transposições ocorrem localmente, ou seja, dentro de pequenas sub-árvores da árvore de busca global. O segundo vetor da tabela de transposição TTABLE (ENTRY* *e2*) utiliza esse esquema de substituição.

Foi possível constatar que, utilizando os dois esquemas de substituição (*deep e new*), os erros tipo 2 foram totalmente controlados no *VisionDraughts*.

Estrutura TTABLE - Manipulação de Dados na Tabela de Transposição com Tratamento de Colisões

Conforme dito anteriormente, a tabela de transposição utilizada pelo *VisionDraughts* é uma estrutura do tipo *TTABLE* composta por dois vetores, (*e1* e *e2* com elementos do tipo *ENTRY*); um método para armazenar novas entradas na tabela; um método para ler as entradas já existentes na tabela.

Em pseudo-código a estrutura de *TTABLE* é a seguinte:

```
class TTABLE{
   int          tableSize;
   ENTRY*       e1;
   ENTRY*       e2;

   StoreEntry (...);
   GetEntry   (...);
}
```

O campo *tableSize* passa a ideia do espaço alocado na tabela de transposição para manipulação de registros do tipo *ENTRY* na memória. Importante notar que *TTABLE* utiliza tanta memória quanta tiver disponível e, claro, quanto mais memória, melhor o desempenho da tabela de transposição.

Cada par de vetor *e1* e *e2* é formado por elementos do tipo *ENTRY* e disponibiliza, em memória, os dados relevantes de até dois estados do tabuleiro do jogo de Damas. Tais dados são obtidos durante o procedimento de busca e os detalhes de como são armazenados e recuperados são definidos, respectivamente, pelos métodos *GetEntry(...)* e *StoreEntry(...)* apresentados a seguir.

$$GetEntry(n; hashvalue; checksum; pdepth); \tag{4.4}$$

O método *GetEntry*(...) da expressão 4.4 visa recuperar informações sobre um dado estado do tabuleiro eventualmente armazenado na tabela de transposição. Para tanto, ele recebe 4 parâmetros de entrada associados ao estado, que são: n representa o próprio estado; *hashvalue* representa a primeira chave *hash* associada ao estado; *checksum* representa a segunda chave hash associada ao estado; e *pDepth* especifica a profundidade mínima de busca desejada.

A partir daí, *GetEntry* localiza o endereço E_1 associado ao parâmetro *hashvalue* na tabela de transposição e verifica se existe algum elemento do tipo *ENTRY* gravado no primeiro vetor e_1 de E_1. Caso exista, verifica se a primeira chave *hash* associada ao elemento de e_1 é igual a *hashvalue*. Caso afirmativo, verifica se a segunda chave *hash* associada ao elemento de e_1 é igual a *checksum*. Caso seja, verifica se a profundidade de busca associada ao elemento de e_1 é maior ou igual a *pDepth*. Caso isso também se confirme, o método *GetEntry* terá obtido sucesso em sua busca e terá encontrado, em *TTABLE*, as informações desejadas para o tabuleiro n. Assim, o algoritmo alfa-beta pode utilizar os dados armazenados em memória em vez de continuar expandido a árvore de busca do jogo. Caso não se tenha obtido êxito com o primeiro vetor e_1, o mesmo processo é realizado, novamente, para o segundo vetor e_2 de E_1.

$$StoreEntry(newEntry); \tag{4.5}$$

O método *StoreEntry(newEntry)*, representado na expressão 4.5, representa o método de armazenamento de informações sobre um dado estado do tabuleiro na tabela de transposição. Para tanto, o método recebe um elemento *newEntry* do tipo *ENTRY* e tenta armazená-lo em *TTABLE*. O método localiza o endereço E_1 em *TTABLE* associado ao parâmetro *hashvalue* de *newEntry*. Considerando o endereço E_1, três situações podem ocorrer:

1. E_1 encontra-se vazio: como inexiste informação gravada no endereço E_1, basta armazenar *newEntry* no primeiro vetor e_1 do endereço E_1;

2. E_1 possui o mesmo elemento passado como parâmetro: o endereço E_1 já possui informação e o valor da primeira chave do primeiro vetor e_1 de E1 é exatamente igual ao valor *hashvalue* de *newEntry*. Assim, caso a profundidade de busca do elemento *newEntry* seja maior que a profundidade de busca do elemento e_1 presente em E_1, transfere-se o elemento presente no vetor e_1 para a mesma posição no vetor e_2 e sobrescreve-se a informação presente em e_1 com a informação de *newEntry*. Caso a profundidade de busca do elemento *newEntry* seja menor, mantém-se a informação presente em e_1 e perdem-se as informações de *newEntry*;
3. E_1 possui elemento diferente: o endereço E_1 já possui informação e o valor da primeira chave do primeiro vetor e_1 de E_1 é diferente do valor *hashvalue* de *newEntry*. Nesse caso, *StoreEntry* resolve a colisão mantendo a informação de e_1 e, independentemente do conteúdo do segundo vetor e_2, gravando o valor de *newEntry* em e_2 ;

Portanto, havendo uma tabela de transposição como a *TTABLE*, sempre que um determinado estado do tabuleiro for apresentado ao algoritmo alfa-beta, ele verificará, primeiro, a tabela de transposição para ver se aquele estado do tabuleiro já foi analisado. Caso afirmativo, a informação armazenada na memória será utilizada diretamente. Caso contrário, a árvore do jogo será expandida pela rotina de busca e uma nova entrada será gravada na tabela de transposição.

4.6.3 Integração entre o Algoritmo Alfa-Beta e a Tabela de Transposição

A primeira tentativa do *VisionDraughts* para integrar o algoritmo alfa-beta com uma tabela de transposição não obteve êxito. A tentativa baseou-se no pseudo-código da seção 4.6.1 e na tabela de transposição da seção 4.6.2. A introdução da estrutura *TTABLE* dentro do procedimento de busca fez surgir erros difíceis de serem rastreados, impedindo a escolha apropriada da melhor ação a ser executada pelo agente jogador, porque a resposta obtida, em alguns casos, não coincidia com a resposta apontada pelo minimax na mesma situação). O problema foi solucionado com a utilização de uma variante do algoritmo alfa-beta chamada *fail-soft* alfa-beta [74], [55].

A Variante Fail-Soft do Algoritmo Alfa-Beta

A variante *fail-soft* alfa-beta é bastante parecida com a *hard-soft*, bastando realizar alterações mínimas na versão *hard-soft* para que se consiga a versão *fail-soft*. Tais alterações, no entanto, provocam grandes mudanças nas predições retornadas pelo algoritmo e permitem a integração com as tabelas de transposição.

O Pseudo-código abaixo apresenta os pontos em que a versão *fail-soft* se difere da versão *hard-soft*:

Pseudo-código do algoritmo alfa-beta *fail-soft*

```
1: alfaBeta(node:n, int:depth, int:min, int:max, move:bestmove)
2: if leaf(n) or depth=0 then
3:    return evaluate(n)
4: end if
5: if n is a max node then
6:    besteval := min
7:    for each child of n: do
8:       v := alphaBeta(child,d-1,besteval,max,bestmove)
9:       if v > besteval then
10:         besteval:= v
11:         thebest = bestmove
12:      end if
13:   end for
14:   if besteval >= max then
15:      return besteval
16:   end if
17:   bestmove = thebest
18:   return besteval
19: end if
20: if n is a min node then
21:   besteval := max
22:   for each child of n: do
23:      v := alphaBeta(child,d-1,min,besteval,bestmove)
24:      if v < besteval then
25:         besteval:= v
26:         thebest = bestmove
27:      end if
28:   end for
29:   if besteval <= min then
30:      return besteval
31:   end if
32:   bestmove = thebest
33:   return besteval
34: end if
```

Sendo que:

- nas **linhas de 14 a 16** se acontecer de a predição armazenada na variável *besteval* ultrapassar o limite superior do intervalo de busca ($besteval \geq max$), o algoritmo retornará, imediatamente, o valor constante da variável *besteval*, em vez do limite superior do intervalo de busca (max), como predição associada ao estado do tabuleiro n. Tal fato expressa a ideia de que a predição associada ao estado do tabuleiro n é, no mínimo, *besteval* ;
- nas **linhas de 29 a 31** se acontecer de a predição armazenada na variável *besteval* for menor que o limite inferior do intervalo de busca ($besteval \leq min$), o algoritmo retornará, imediatamente, o valor constante da variável *besteval*, em vez do limite inferior do intervalo de busca (min), como predição associada ao estado do tabuleiro n. Tal fato expressa a ideia de que a predição associada ao estado do tabuleiro n é, no máximo, *besteval*;

Com base na explicação do algoritmo *fail-soft* alfa-beta, mostrada acima, pode-se concluir que:

- na **poda alfa**: a avaliação dos filhos de um nó n de minimização pode ser interrompida tão logo uma predição P_i, calculada para um dos filhos de n, seja menor que o parâmetro alfa. Nesse caso, o valor P_i indica que o nó n possui predição igual a, no máximo, P_i;
- na **poda beta**: a avaliação dos filhos de um nó n de maximização pode ser interrompida tão logo uma predição P_i, calculada para um dos filhos de n, seja maior que o parâmetro beta. Nesse caso, o valor P_i indica que o nó n possui predição igual a, no mínimo, P_i.

Tanto a versão *hard-soft* quanto a *fail-soft* retornam a mesma predição para um dado estado, ambas efetuam as mesmas podas na árvore de busca, porém a versão *fail-soft* armazena o valor real da predição do estado e não o valor do limite (*min ou max*) que efetuou a poda na busca, como é feito na versão *hard-soft*. Portanto, durante a construção do *VisionDraughts*, foi possível concluir que a versão *hard-soft* é incompatível com o uso de tabelas de transposição: caso, com a evolução do jogo, um dado estado S_i precise ser avaliado novamente, pode acontecer de um valor indevido de S_i ser resgatado da tabela de transposição. A versão *fail-soft* garante que caso o estado S_i esteja na tabela, seu valor será real, não se tratando de um valor de delimitador de busca.

A seguir, serão analisados os métodos utilizados para ler e escrever dados na tabela de transposição a partir do algoritmo *fail-soft alfa-beta*.

Armazenar Estados do Tabuleiro na Tabela de Transposição a partir do Algoritmo Fail-Soft Alfa-Beta

Como o algoritmo alfa-beta não mantém um histórico dos estados da árvore de jogo procurados anteriormente, se um estado do tabuleiro for apresentado 2 vezes para o algoritmo alfa-beta, a mesma rotina será executada 2 vezes a fim de encontrar a predição associada ao estado. A tabela de transposição utilizada pelo *VisionDraughts* evita a referida pesquisa em duplicidade, armazenando e recuperando dados em memória.

A estrutura *ENTRY*, mostrada na seção 4.6.2, é a unidade básica de entrada para a tabela de transposição utilizada pelo *VisionDraughts*. Um método para armazenar estados do tabuleiro na tabela de transposição precisa criar uma nova estrutura do tipo *ENTRY* e armazená-la no endereço de memória correto. Então, para que seja possível criar uma estrutura do tipo *ENTRY*, o método precisa ter a seguinte assinatura:

$$store(n;\ besteval;\ bestmove;\ depth;\ scoreType); \tag{4.6}$$

onde n representa um estado qualquer do tabuleiro do jogo, *besteval* representa a predição associada ao estado n, *bestmove* corresponde ao melhor movimento para o jogador a partir de n, *depth* representa a profundidade de busca associada a n e *scoreType* indica se a predição do estado do tabuleiro n é exatamente igual a *besteval*, no máximo, igual a *besteval*, ou, no mínimo, igual a *besteval*. Considerando que todo e qualquer estado do tabuleiro do jogo possui duas chaves hash associadas (seção 4.6.2), o método acima já possui todas as informações para escrever na tabela de transposição, bastando obedecer o esquema de substituição descrito na seção 4.6.2. Para tanto, o método *store* invoca o método *StoreEntry* (equação 4.5).

O detalhe mais importante do método *store* é o parâmetro *scoreType*. Ele pode assumir os valores *hashExact*, *hashAtLeast* ou *hashAtMost* de acordo com a seguintes regras:

1. **ausência de poda**: caso o estado do tabuleiro n seja uma folha da árvore de busca ou caso todos os F_i filhos de n tenham sido analisados (não ocorre poda), significa que a predição associada a n é exatamente igual a *besteval*. Então, scoreType deve receber o valor *hashExact*;
2. **poda alfa**: caso algum dos filhos F_i de n tenha sido descartado do procedimento de busca por uma poda alfa, significa que n é um minimizador e sua predição é, no máximo, igual a *besteval*. Então, *scoreType* deve receber o valor *hashAtMost*;
3. **poda beta**: caso algum dos filhos F_i de n tenha sido descartado do procedimento de busca por uma poda beta, significa que n é um maximizador e sua predição é, no mínimo, igual a *besteval*. Então, *scoreType* deve receber o valor *hashAtLeast*.

Quando acontece algum tipo de poda nos filhos de n tem-se que, a predição associada a n é igual a *hashAtMost* no caso de poda alfa, ou igual a *hashAtLeast* no caso de poda

beta, como explicado anteriormente. Para saber que valor está associado a n na tabela de transposição quando ocorre alguma poda, deve-se considerar que:

1. um nó n armazenado com predição P_i e *scoreType* = *hashAtMost* indica que:

   - o nó n foi avaliado em nível de minimização da árvore de busca e, em tal avaliação, ocorreu uma poda alfa. O pai de n estava em nível de maximização e a avaliação de n provoca reajustes de incremento no limite inferior do intervalo de busca do pai de n;
   - considerando que n tenha k filhos com predições $P_1, ..., P_k$, então $P_i \epsilon \{P_1, ..., P_k\}$, isto é, P_i é a predição de um dos filhos F_i de n;
   - a predição dos nós podados (irmãos de F_i) $\epsilon \{P_j \mid 1 \leq j \leq k \text{ e } j \neq i\}$;
   - $P_i < \alpha$ (alfa), onde alfa era o limite inferior do intervalo de busca no momento em que foi calculada a predição P_i para n.

2. um nó n armazenado com predição P_i e *scoreType* = *hashAtLeast* indica que:

   - o nó n foi avaliado em nível de maximização da árvore de busca e, em tal avaliação, ocorreu uma poda beta. O pai de n estava em nível de minimização e a avaliação de n provoca reajustes de decremento no limite superior do intervalo de busca do pai de n;
   - considerando que n tenha k filhos com predições $P_1, ..., P_k$, então $P_i \epsilon \{P_1, ..., P_k\}$, isto é, P_i é a predição de um dos filhos F_i de n;
   - a predição dos nós podados (irmãos de F_i) $\epsilon \{P_j \mid 1 \leq j \leq k \text{ e } j \neq i\}$;
   - $P_i > \beta$ (beta), onde beta era o limite superior do intervalo de busca no momento em que foi calculada a predição P_i para n.

Nas próximas seções, quando forem encontradas as expressões *hashExact*, *hashAtMost* ou *hashAtLeast* no que diz respeito às predições dos estados de tabuleiros, considere que essas predições foram obtidas seguindo os critérios apresentados acima.

Recuperação dos Estados do Tabuleiro da Tabela de Transposição a partir do Algoritmo Fail-Soft Alfa-Beta

O *VisionDraughts* tenta utilizar os dados armazenados em memória em vez de expandir uma árvore de busca do jogo. Um método para verificar se um determinado estado do tabuleiro do jogo encontra-se armazenado na tabela de transposição precisa ter a seguinte assinatura:

retrieve(n; besteval; bestmove; depth; nodeType); (4.7)

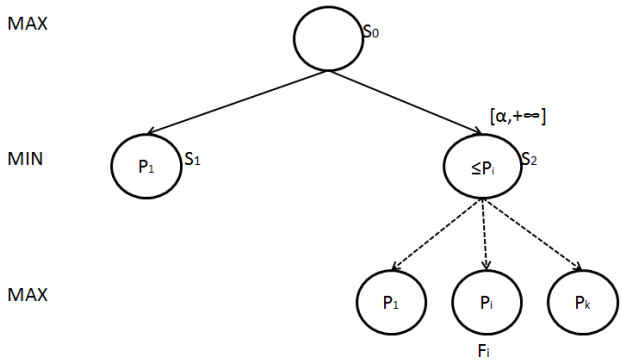

Figura 4.9: Árvore de busca onde o pai de n é maximizador

O estado do tabuleiro do jogo que está sendo procurado na tabela de transposição é representado por n, *depth* representa a profundidade de busca associada a n, *nodeType* indica se o estado pai de n é **minimizador** ou **maximizador**, *besteval* e *bestmove* são parâmetros de saída que indicarão a predição e a melhor jogada associadas ao estado n, respectivamente, caso ocorra sucesso no procedimento de recuperação do estado n na tabela de transposição.

Assim sendo, inicialmente, o método *retrieve* aciona o procedimento de leitura *GetEntry* (4.4) para checar se o estado n está armazenado na tabela de transposição. Caso obtenha sucesso e consiga recuperar uma entrada E_1 na tabela correspondente ao estado n, tal entrada precisa ser tratada de acordo com o parâmetro *nodeType*, a fim de verificar se os valores constantes da tabela de transposição podem ser utilizados para preencher os parâmetros de saída *besteval* e *bestmove*.

O estado do tabuleiro n no método *retrieve* representa, sempre, um nó filho. O tratamento baseado no parâmetro *nodeType* é feito levando-se em conta o fato de o pai de n ser maximizador ou minimizador.

Considere o exemplo da figura 4.9 para verificar se os valores da tabela de transposição podem ser utilizados para preencher os parâmetros de saída *besteval* e *bestmove* levando em consideração se o pai de n é **maximizador**, sendo que:

- a predição de S_1 está na tabela da transposição com valor exato de predição (isto é, *hashExact*) iguais a P_1. S_2 está na tabela de transposição com um valor de predição *hashAtMost* P_i;
- suponha que S_2 tenha k filhos (parte pontilhada na figura 4.9).

Quando se que o pai de n, representado por S_0, é um nó maximizador, algum dos valores de predição P_1 e P_i associados aos estados S_1 e S_2, respectivamente, fornecidos pela tabela de transposição pode ser usado?

Partindo da premissa de que a profundidade de busca associada a cada uma das predições P_1 e P_i seja igual ou maior que a profundidade de busca dos estados S_1 e S_2 pode-se concluir que:

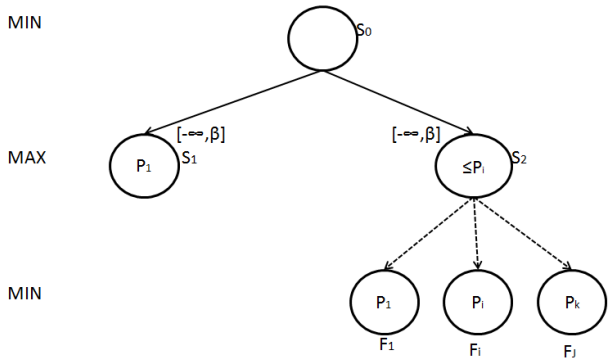

Figura 4.10: Árvore de busca onde o pai de n é minimizador

1. o valor de P_1 pode ser usado desde que o valor *hashExact* tenha sido obtido no nível de minimização, ou seja, no mesmo nível de n. Caso tenha sido obtido em nível de maximização (nível oposto ao de n), o valor de P_1 será o oposto do valor de predição desejado pelo nível de minimização;
2. P_i somente pode ser usado se $P_i \leq \alpha$, onde α é o limite inferior do intervalo de busca com que S_2 seria expandido. Note que S_2, sempre, será podado pelo limite de maximização de S_0. De fato, nesse caso, nenhum filho de n com predição menor que P_i "subiria"para S_0, pois o valor de α impediria. Analogamente, nenhum filho de n com predição maior que P_i "subiria"para S_0 pois o minimizador de S_2 impediria;
3. caso nenhum dos itens acima possam ser usados, os estados S_1 e S_2 precisam ser expandidos.

Para a análise de quando o pai de n está em nível de minimização, considere a árvore da figura 4.10:

Partindo da premissa de que a profundidade de busca associada a cada uma das predições P_1 e P_i seja igual ou maior que a profundidade de busca dos estados S_1 e S_2, pode-se concluir que:

1. o valor de P_1 pode ser usado desde que o valor *hashExact* tenha sido obtido no nível de maximização, ou seja, no mesmo nível de n. Caso tenha sido obtido em nível de minimização (nível oposto ao de n), o valor de P_1 será o oposto do valor de predição desejado pelo nível de maximização;
2. se $P_i \geq \beta$, o valor associado a P_i pode ser usado, uma vez que n será descartado, pois:
   - todo irmão F_j de F_i que tiver sido podado na poda Beta que gerou F_i, e que tiver predição $P_j > P_i$ "subirá"pelo maximizador de n, mas será barrado pelo minimizador de β;
   - todo irmão F_j de F_i que tiver sido podado na poda Beta que gerou F_i, e que tiver predição $P_j < P_i$ será descartado pelo maximizador de n.

3. se $P_i < \beta$, o valor associado a P_i não pode ser usado, pois todo irmão F_j de F_i que tiver sido podado na poda beta que gerou P_i, e que tiver predição P_j, tal que $P_i < P_j < \beta$ seria candidato a "subir"até S_0. Inicialmente "subiria"até n caso P_j fosse maior que as predições dos filhos de n, posteriormente, "subiria"até S_0, caso P_j fosse a menor entre as predições dos filhos de S_0;
4. caso nenhum dos itens acima possam ser usados os estados S_1 e S_2 precisam ser expandidos.

O Algoritmo Fail-Soft Alfa-Beta com Tabela de Transposição

Todos os pré-requisitos para a integração entre algoritmo alfa-beta e tabelas de transposição, no jogo de Damas, foram detalhados nas seções anteriores. Para esclarecer o assunto, definitivamente, será apresentado o pseudo-código do algoritmo utilizado pelo *VisionDraughts* e detalhadas as linhas mais importantes (as linhas que diferem do pseudo-código da variante *fail-soft* alfa-beta sem tabela de transposição, mostrados na seção 4.6.3).

Pseudo-código do algoritmo *fail-soft* alpha-beta com tabela de transposição

```
1: alfaBeta(node:n, int:depth, int:min, int:max, move:bestmove)
2: if leaf(n) or depth=0 then
3:    besteval := evaluate(n)
4:    store(n, besteval,bestmove,depth,hashExact)
5:    return besteval
6: end if
7: if n is a max node then
8:    besteval := min
9:    for each child of n do
10:      if retrieve(child,besteval,bestmove,depth-1,parentIsMaxNode) then
11:         then v := besteval
12:      else
13:         v := alfabeta(child,depth-1,besteval,max,bestmove)
14:      end if
15:      if v > besteval then
16:         besteval:= v
17:         thebest = bestmove
18:      end if
19:      if besteval >= max then
20:         store(child,besteval,bestmove,depth,hashAtLeast)
21:         return besteval
```

```
22:       end if
23:     end for
24:     bestmove = thebest
25:     store(n,besteval,bestmove,depth,hashExact)
26:     return besteval
27: end if
28: if n is a min node then
29:     besteval := max
30:     for each child of n do
31:       if retrieve(child,besteval,bestmove,depth-1,parentIsMinNode) then
32:         then v := besteval
33:       else
34:         v := alfabeta(child,depth-1,besteval,max,bestmove)
35:       end if
36:       if v < besteval then
37:         besteval:= v
38:         thebest = bestmove
39:       end if
40:       if besteval <= min then
41:         store(child,besteval,depth,hashAtMost)
42:         return besteval
43:       end if
44:     end for
45:     bestmove = thebest
46:     store(n, besteval,depth,hashExact)
47:     return besteval
48: end if
```

- **linha 4**: quando o algoritmo de busca chamar a rede neural na linha 3, com a rotina *evaluate(n)*, para conhecer a predição associada ao estado folha n, o valor da predição recém calculada deverá ser armazenado na tabela de transposição. Assim, o algoritmo invoca o procedimento *store* (4.6), descrito na seção 4.6.2. Como n é um nó folha, o valor da predição deve ser armazenado com *scoretype* = *hashExact*, ou seja, predição exatamente igual a *besteval*;
- **linha 10**: para cada um dos filhos de um estado maximizador n, antes de chamar a rotina de busca recursivamente, o repositório de dados em memória (tabela de transposição) deverá ser consultado. Então o algoritmo invoca o procedimento *retrieve* (4.7), também descrito na seção 4.6.2, com *nodeType* = *parentIsMaxNode*. Outro detalhe muito importante é que, nesta linha, a profundidade de busca deve

ser igual a depth - 1, que corresponde à profundidade de n;

- **linha 11**: se o filho do estado n estiver armazenado na tabela de transposição e o método *retrieve* obtiver sucesso no tratamento das informações, o valor da predição retornado por *retrieve* deverá ser utilizado, em vez de chamar o algoritmo alfa-beta recursivamente. Veja que o método *retrieve* (4.7)é chamado com um *flag* indicando que o estado do tabuleiro n é do tipo maximizador;
- **linha 20**: Detectada a ocorrência de uma *poda beta*, quando o algoritmo estiver analisando os filhos de um nó n do tipo maximizador, significará que os demais filhos de n, caso existam, não precisam mais ser analisados. A variável *besteval* conterá, nesta linha, o valor mínimo aceitável para o estado do tabuleiro representado pelo nó n. Assim, a predição presente na variável *besteval* deverá ser armazenada na tabela de transposição pelo método *store* (4.6) com o *flag hashAtLeast* indicando que a predição associada ao estado do tabuleiro n é, no mínimo, igual a *besteval* ;
- **linha 25**: quando o algoritmo tiver analisado todos os filhos de um nó n do tipo maximizador, a variável *besteval* conterá exatamente o valor da predição para o estado do tabuleiro representado pelo nó n. Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store* (4.6) com o *flag hashExact* que indica que a predição associada ao estado do tabuleiro n é exatamente igual a *besteval* ;
- **linha 31**: se o filho do estado n estiver armazenado na tabela de transposição e o método *retrieve* obtiver sucesso no tratamento das informações, o valor da predição retornado por *retrieve* deve ser utilizado, em vez de chamar o algoritmo alfa-beta recursivamente. Veja que o método *retrieve* (4.7) é chamado com um *flag* indicando que o estado do tabuleiro n é do tipo minimizador;
- **linha 41**: detectada a ocorrência de uma *poda alfa*, quando o algoritmo estiver analisando os filhos de um nó n do tipo minimizador, significa que os demais filhos de n, caso existam, não precisam mais ser analisados. A variável *besteval* conterá, nesta linha, o valor máximo aceitável para o estado do tabuleiro representado pelo nó n. Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store* (4.6) com o *flag hashAtMost* indicando que a predição associada ao estado do tabuleiro n é, no máximo, igual a *besteval*;
- **linha 46**: quando o algoritmo analisar todos os filhos de um nó n, do tipo minimizador, a variável *besteval* conterá exatamente o valor da predição para o estado do tabuleiro representado pelo nó n. Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store* (4.6) com o *flag hashExact* indicando que a predição associada ao estado do tabuleiro n é exatamente igual a *besteval*.

O segundo avanço do *VisionDraughts* em relação ao sistema *NeuroDraughts* [42], [43] foi conseguido com o uso das tabelas de transposição em conjunto com o *fail-soft* alfa-beta. O uso de tabelas de transposição reduz o tempo de execução necessário para encontrar a melhor ação a ser executada pelo agente jogador para menos de 40% do tempo exigido pelo algoritmo alfa-beta puro.

4.6.4 O Aprofundamento Iterativo

A qualidade de um programa jogador que utiliza o algoritmo de busca alfa-beta depende muito do número de jogadas que ele pode olhar adiante (*look-ahead*). Para jogos com um fator de ramificação grande, o jogador pode levar muito tempo para pesquisar poucos níveis adiante.

Em jogos de Damas, a maioria dos programas utilizam mecanismos para delimitar o tempo máximo permitido de busca. Como o algoritmo alfa-beta realiza uma busca com profundidade fixa, não existe garantia de que a busca irá se completar antes do tempo máximo estabelecido. Para evitar que o tempo se esgote e o programa jogador não possua nenhuma informação de qual a melhor jogada a ser executada, profundidade fixa não deve ser utilizada [45].

Larry Atkin [22] introduziu a técnica de aprofundamento iterativo como um mecanismo de controle do tempo de execução durante a expansão da árvore de busca, conforme ilustrado na figura 4.11. Note que a ideia básica do aprofundamento iterativo é realizar uma série de buscas, em profundidade, independentes, cada uma com um *look-ahead* acrescido de um nível. Assim, é garantido que o procedimento de busca iterativo encontra o caminho mais curto para a solução justamente como a busca em largura encontraria. Porém, em comparação com a última estratégia, os recursos de memória são insignificantes. Em outras palavras, uma busca com aprofundamento iterativo "imita"uma busca em largura com uma série de buscas em profundidade [45].

Seguindo a ideia de Atkin, o *VisionDraughts* utiliza a técnica de aprofundamento iterativo da seguinte forma: inicialmente, o algoritmo alfa-beta pesquisa com profundidade 2, depois com profundidade 4, depois com profundidade 6 e assim, sucessivamente, até que o tempo máximo de busca se esgote.

No *VisionDraughts*, o algoritmo alfa-beta é chamado com profundidades *depth* = 2, 4, 6, ..., max, até que o tempo de busca se esgote em uma profundidade qualquer *depth* = *d*, tal que $4 \leq d \leq max$. Nesta situação, os resultados encontrados para a profundidade *depth* = *d - 2* são utilizados (se o tempo de busca não se esgotar, os resultados encontrados para a profundidade *depth* = *max* são utilizados). Conforme dito na seção 2.5.2, a desvantagem do aprofundamento iterativo é o processamento repetido de estados de níveis mais rasos da árvore de busca. Para atenuar tal inconveniente, o *VisionDraughts* lança mão de repositório da tabela de transposição descrito na seção anterior.

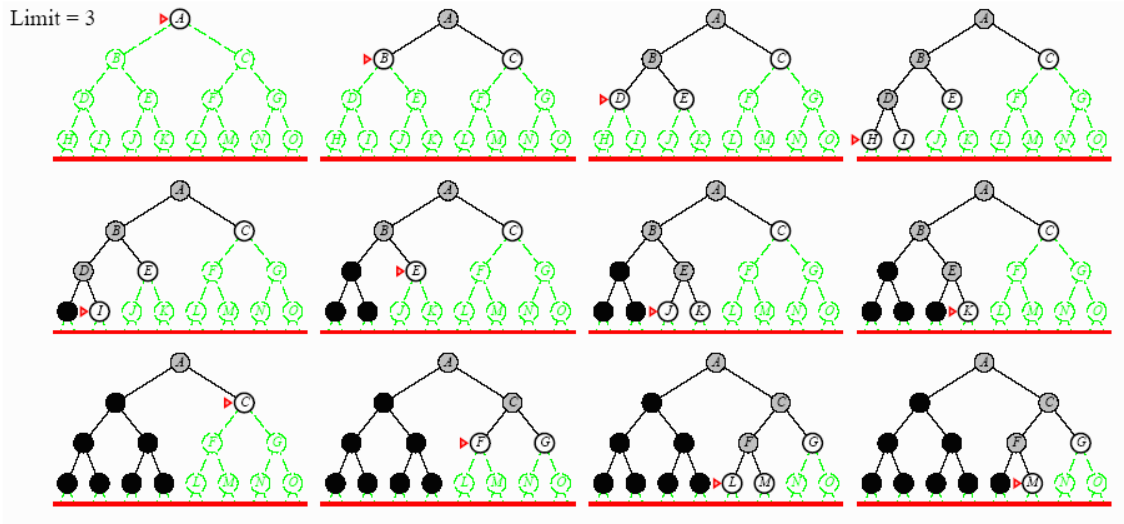

Figura 4.11: Árvore de busca criada pelo Aprofundamento Iterativo com limite de profundidade 3

Convém salientar que o *MP-Draughts*, além do recurso de tabela de transposição, também utiliza uma estratégia de ordenação parcial para atacar o problema de redundância de expansão de estados ocasionada pelo aprofundamento iterativo, conforme apresentado no capítulo 5.

4.6.5 O Algoritmo Alfa-Beta com a Tabela de Transposição e o Aprofundamento Iterativo

Tendo integrado o algoritmo alfa-beta com a tabela de transposição, falta somente a integração com o aprofundamento iterativo para que o eficiente módulo de busca utilizado pelo *VisionDraughts* fique completo.

A busca com aprofundamento iterativo é, normalmente, mais eficiente que uma busca alfa-beta com profundidade fixa, além dissom proporciona um bom controle de tempo. A visualização da eficiência do procedimento iterativo não é óbvio e, de fato, o procedimento não é eficiente se não for utilizado em conjunto com tabelas de transposição.

Para executar o procedimento alfa-beta repetidas vezes, ou seja, introduzir aprofundamento iterativo, é necessário uma rotina de iteração. Essa rotina usará o algoritmo *fail-soft* alfa-beta com tabela de transposição, descrito acima. O pseudo-código da rotina pode ser visualizado a seguir:

Rotina de iteração

```
1: iterative(node:n, int:depth, int:min, int:max, move:bestmove)
2: besteval = alfaBeta(n, 2, min, max, bestmove)
3: start = clock()
4: for d=4 ; d ≤ MAX-SEARCH-DEPTH ; d=d+2 do
5:     besteval = alfaBeta(n, d, min, max, bestmove)
```

```
6:    stop = clock()
7:    if (stop - start) > MAX-EXECUTION-TIME then
8:       return besteval
9:    end if
10: end for
11: return besteval
```

Abaixo segue a explicação em detalhes do como tal rotina funciona:

- **linha 1**: a rotina de iteração possui os mesmos parâmetros utilizados no procedimento de busca alfa-beta com tabela de transposição;
- **linha 2**: o primeiro passo de iteração é realizado com profundidade igual a 2. O melhor movimento a ser realizado é armazenado em *bestmove* e a predição em *besteval*;
- **linha 3**: a variável *start* representa o início da contagem de tempo disponível para que o agente jogador encontre o melhor movimento a ser realizado;
- **linha 4**: o procedimento iterativo se repetirá até que o limite de tempo ou profundidade de busca máxima seja alcançada;
- **linha 5**: representa os passos de iteração com profundidade $depth \geq 4$;
- **Linha 6**: a variável *stop* representa o fim cronológico de um passo de iteração;
- **linha 8**: se o tempo máximo estabelecido para que o agente jogador encontre o melhor movimento a ser realizado se esgotar, o procedimento iterativo deve ser encerrado com retorno do melhor movimento *besteval* encontrado até o presente momento;
- **linha 11**: retorna o melhor movimento *besteval* após a profundidade máxima de busca ter sido atingida.

O terceiro avanço do *VisionDraughts* em relação ao sistema *NeuroDraughts* é conseguido com o uso da rotina de aprofundamento iterativo. Tal rotina contribui parcialmente para a diminuição da ocorrência do problema do *loop*, identificado por Neto e Julia [50], [51], [52] no trabalho de Lynch [42]. O problema do *loop* representa a situação na qual o agente jogador de Damas, apesar de estar em vantagem com relação a seu adversário, não consegue pressioná-lo e entra em *loop* infinito, não evoluindo a partida. O aprofundamento iterativo contribui parcialmente com o problema do *loop*, pois em algumas situações o jogador consegue visualizar jogadas diferentes com um *look-ahead* maior e variável.

4.7 Análise em Retrocesso e Base de Dados

O uso de bases de dados para as fases finais do jogo de Damas tem papel fundamental na construção de agentes jogadores de alto nível, pois embora o estado do tabuleiro presente na raiz da árvore de busca possa estar longe do fim do jogo, os estados representados pelas folhas já podem estar nas bases de dados [40].

Essencialmente, a construção de bases de dados para fases finais do jogo é realizada por uma "busca para trás"partindo da solução do problema. Tal técnica, combinada com o algoritmo alfa-beta que realiza uma "busca para frente", permite que se encontre a solução ótima para o problema de escolha da melhor ação em menos tempo. O grande sucesso do programa *Chinook*, por exemplo, se baseia principalmente em suas bases de dados de finais de jogos [40].

O espaço de busca do jogo de Damas possui aproximadamente 5 x 10^{20} estados distintos do tabuleiro. Sendo assim, a construção de bases de dados para as fases finais do jogo é uma tarefa difícil. No projeto *Chinook* os esforços para construção das bases de dados se iniciaram em 1989 e, desde então, dezenas de computadores trabalham exclusivamente com esse intuito [67]. Em 1992 existiam mais de 200 computadores trabalhando simultaneamente na construção das bases de dados [64].

Considerando a dificuldade de construção de bases de dados próprias, o *VisionDraughts* utiliza as amostras disponibilizadas para download pela equipe do *Chinook* [69]. Tais amostras possuem mais 5.0 GB de tamanho e informação perfeita (vitória, derrota ou empate) para os seguintes estados do tabuleiro envolvendo 8 peças ou menos:

1. todos os estados do tabuleiro com 6 peças ou menos;
2. os estados do tabuleiro formados pela combinação 4 x 3 peças ;
3. os estados do tabuleiro formados pela combinação 4 x 4 peças ;

Apesar de o *VisionDraughts* utilizar apenas uma amostra das bases de dados de 8 peças ou menos, seu desempenho é melhorado, consideravelmente, em dois sentidos: - quando um estado do tabuleiro é encontrado na base de dados, o *VisionDraughts* utiliza a predição exata para ele (vitória, empate ou derrota) em vez de utilizar a função de avaliação heurística; - quando um estado do tabuleiro é encontrado em uma base de dados, o *VisionDraughts* não precisa pesquisar nenhum dos descendentes dele na árvore de busca.

O efeito combinado dos benefícios acima resulta em um mecanismo de busca que expande uma árvore de busca menor e alcança resultados mais precisos [64], uma vez que mais estados do tabuleiro serão associados com predições exatas (vitória, derrota ou empate) extraídas das bases de dados.

Sendo assim, de posse das bases de dados, resta detalhar os procedimentos utilizados pelo *VisionDraughts* para consultar se um determinado estado do tabuleiro encontra-se

presente nas bases e como utilizá-las em conjunto com o algoritmo *fail-soft* alfa-beta com tabelas de transposição.

Para fins de integração com o procedimento alfa-beta, as seguintes linhas são acrescentadas ao algoritmo mostrado na seção 4.6.3.

Pseudo-código do algoritmo fail-soft alpha-beta com tabela de transposição e bases de dados de finais de jogos

```
1: alfaBeta(node:n, int:depth, int:min, int:max, move:bestmove)
2: if (not isRoot(n)) and (isLookupBoard(n)) then
3:    getBitBoards(n, BP,WP,K)
4:    db-value = lookup-positions(..., BP, WP, K)
5: end if
6: if (db-value==1) and (n is a min node) then
7:    return -1.0
8: end if
9: if (db-value==1) and (n is a max node) then
10:   return +1.0;
11: end if
12: if (db-value==2) and (n is a min node) then
13:   return +1.0;
14: end if
15: if (db-value==2) and (n is a max node) then
16:   return -1.0;
17: end if
18: if (db-value==3) then
19:   return 0.0;
20: end if
21: if leaf(n) or depth=0 then then
22:   ...
23:   ...
24: end if
25: if  n is a max node then
26:   ...
27:   ...
28: end if
29: if  n is a min node then
30:   ...
31:   ...
32: end if
```

1. **linha 2**: a função *isLookupBoard* é utilizada no sentido de garantir que o tabuleiro do jogo cumpra os pré-requisitos necessários para que possa ser consultado junto às bases de dados. A função *isRoot* é utilizada para garantir que os estados consultados nas bases de dados sempre tenham, no mínimo, um antecessor na árvore de busca (todos os estados da árvore com exceção de sua raiz);
2. **linha 3**: a função *getBitBoards* pega um estado do tabuleiro e gera os *BitBoards* (BP, WP, K) necessários para integração com as bases de dados. *BitBoards* é o modo usado pelo *Chinook* para representar tabuleiro de jogo de Damas;
3. **linha 4**: a função *lookup-positions* realiza o procedimento de consulta junto às bases de dados;
4. **linha 6**: o resultado *db-value* = 1 indica que o estado do tabuleiro n representa vitória para o próximo jogador a se mover. Logo, o estado do tabuleiro n representa derrota para o estado pai de n. Depois de todas essas considerações, é possível concluir que o valor -1.0 deve ser retornado na linha 7 para indicar que o pai de n (maximizador) encontra-se derrotado na partida de acordo com as bases de dados;
5. **linha 9**: o resultado *db-value* = 1 indica que o estado do tabuleiro n representa vitória para o próximo jogador a se mover. Logo, o estado do tabuleiro n representa derrota para o estado pai de n. É possível concluir que o valor +1.0 deve ser retornado na linha 10 para indicar que o pai de n (minimizador) encontra-se derrotado na partida de acordo com as bases de dados;
6. **linha 12**: o resultado *db-value* = 2 indica que o estado do tabuleiro n representa derrota para o próximo jogador a se mover. Logo, o estado do tabuleiro n representa vitória para o estado pai de n. É possível concluir que o valor +1.0 deve ser retornado na linha 13 para indicar que o pai de n (maximizador) encontra-se vitorioso na partida de acordo com as bases de dados;
7. **linha 15**: o resultado *db-value* = 2 indica que o estado do tabuleiro n representa derrota para o próximo jogador a se mover. Logo, o estado do tabuleiro n representa vitória para o estado pai de n. É possível concluir que o valor -1.0 deve ser retornado na linha 16 para indicar que o pai de n (minimizador) encontra-se vitorioso na partida de acordo com as bases de dados;
8. **linha 18**: O resultado *db-value* = 3 indica que o estado do tabuleiro n representa empate para o próximo jogador a se mover. Logo, o estado do tabuleiro n representa empate para o estado pai de n. Portanto, o valor 0.0 deve ser retornado na linha 19 para indicar que o pai de n encontra-se empatado na partida de acordo com as bases de dados;

Importante ressaltar que os valores - 1.0, 0.0 e + 1.0 podem ser substituídos por outros que tendem a eles. Por exemplo, em vez de utilizar + 1.0 representando vitória para um

estado maximizador, pode-se utilizar um valor x, tal que x seja quase + 1.0. Fazendo isso, o valor + 1.0 passa a representar somente as vitórias expressas diretamente no tabuleiro e o valor x passa a representar as vitórias expressas nas bases de dados.

A inserção dos dois módulos de busca no *VisionDraughts* (base de dados e algoritmo de busca eficiente) diminuiu consideravelmente a ocorrência de *loops* nesse jogador. O problema de frequente ocorrência de *loop* é um dos fatores negativos na performance do jogador *NeuroDraughts*, predecessor do *VisionDraughts*. A simples utilização de uma maior profundidade de busca (look-ahead) já forneceu ao *VisionDraughts* capacidade de superar o nível de jogo do *NeuroDraughts*.

Ao leitor de desejar conhecer melhor todos os detalhes do jogador *VisionDraughts*, é sugerido a leitura do trabalho de Caixeta e Julia [11]. Não é foco desse trabalho apresentar detalhadamente o *VisionDraughts*, já que este trabalho tem um objetivo maior que é criar um jogador especialista em final de jogo. Como dito anteriormente, o *VisionDraughts* foi usado como jogador de início e meio de jogo, por ser um bom jogador de Damas e também por fazer parte de um grupo de estudos no qual ambos os jogadores estão envolvidos.

Capítulo 5

MP-Draughts - Um Sistema Muti-Agente de Aprendizado de Damas

Neste capítulo, será apresentado o *MP-Draughts* (*MultiPhase Draughts*): um sistema multi-agente de aprendizagem de jogos de Damas cujo principal objetivo é construir um agente automático capaz de jogar Damas com alto nível de desempenho ao longo de um jogo. A proposta para esse jogador é que o mesmo tenha o mínimo de intervenção humana possível e use somente poderosas técnicas de Inteligência Artificial no seu processo de aprendizagem.

A eficiência do jogador é conseguida por meio de eficientes técnicas de Inteligência Artificial como auxílio no seu processo de aprendizagem, ao contrário do *Chinook* [71] (melhor jogador de Damas automático do mundo), que conta com ajustes manuais em suas funções de aprendizagem e, além disso, usa duas bases de dados para auxiliar na tomada de decisão de quais jogadas executar no início e no final de um jogo. Essas bases de dados foram construídas por especialistas humanos e contêm informações precisas de boas jogadas de início de jogo (*Openning Book*) e informações exatas quanto a jogadas de finalização de jogo. Na base de dados de final de jogo é possível encontrar o resultado de qualquer jogo que tenha dez peças ou menos sobre o tabuleiro, ou seja, nessa base existem informações sobre vitória, empate ou derrota de qualquer configuração de tabuleiro com 10 peças ou menos [71], [70]. Como é possível notar, o melhor jogador do mundo contou com muita intervenção humana para conseguir tal título, e não é esse o objetivo desse trabalho. Relembrando que o objetivo aqui é ter um jogador de Damas competitivo e autônomo (baseado na sua própria capacidade de escolha) para tomar decisões sobre como agir durante um jogo.

O *MP-Draughts* é composto por 26 agentes, sendo que um agente (***IIGA*** - *Initial/Intermidiate Game Agent*) foi construído e treinado para ser especialista em fases

iniciais e intermediárias de jogo, ao passo que os outros 25 agentes(**EGA** - *EndGame Agent*) foram construídos e treinados para serem especialistas em fases de final de jogo (cada um treinado para ser capaz de lidar com um determinado tipo de tabuleiro de final de jogo).

Para facilitar o entendimento do que é fase inicial/intermediária de jogo e fase de final de jogo, desse ponto em diante, a expressão "fase inicial/intermediária de jogo"deve ser entendida como configuração de tabuleiro com, no mínimo, 13 peças. Já a expressão "fase de final de jogo"refere-se a tabuleiros com, no máximo, 12 peças, ao passo que "tabuleiro de final de jogo"referencia tabuleiros com exatas 12 peças.

O jogador multiagente tratado nesse trabalho é fruto de uma contínua atividade de pesquisas que teve como frutos anteriores o jogador de Neto e Julia, o *LS-Draughts* [50] e o jogador de Caixeta e Julia, o *VisionDraughts* [11], sendo que este último foi tratado no capítulo 4. E assim como nesse jogadores, a linguagem de programação usada foi o C++.

Cada agente que compõe o *MP-Draughts* apresenta a mesma arquitetura do *Vision-Draughts*, ou seja, cada agente jogador é implementado como uma rede neural multicamadas (*Multi-Layer Perceptron* - MLP) que utiliza de busca alfa-beta com tabela de transposição e aprofundamento iterativo para a escolha da melhor jogada em função do estado corrente do tabuleiro do jogo. Além disso, utiliza o método de aprendizagem por reforço por Diferenças Temporais TD(λ) aliado à estratégia de treino por *self-play* com clonagem, como ferramentas para atualizar os pesos da rede. Para tanto, o tabuleiro é representado por um conjunto de funções que descrevem as características do próprio jogo de Damas. A utilização de um conjunto de características para representar o mapeamento do tabuleiro de Damas é a mesma definida por Lynch em seu jogador *NeuroDraughts* [42] como sendo um mapeamento *NET-FEATUREMAP*. A representação do conjunto de características usadas no mapeamento dos tabuleiros foi descrita na seção 4.1 desta dissertação.

Apesar do bom desempenho, o *VisionDraughts* (jogador usado como base para o desenvolvimento deste trabalho), assim como outros bons agentes jogadores de Damas que aprendem por reforço, tais como Anaconda [14], [19], *NeuroDraughts* [42], [43], e *LS-Draughts* [50], [51], frequentemente, não terminam com sucesso a fase final do jogo, mesmo estando em situação de vantagem quando comparado com seu oponente. Tal situação de insucesso inclui *loops* de final de jogo. Um *loop* ocorre quando o agente, mesmo em vantagem, não consegue pressionar o adversário e alcançar a vitória. Ao invés disso, o agente começa uma sequência repetitiva de movimentos (*loop*) alternando-se entre posições inúteis do tabuleiro. Tais movimentos não modificam o estado do jogo. O problema de *loop* no *VisionDraughts* é constatado sempre em fase de final de jogo.

Na intenção de atacar esse mal comportamento durante um jogo de Damas, o *MP-Draughts* conta com 25 redes neurais especializadas em final de jogo, cada uma treinada para ser capaz de lidar com um determinado *cluster* de final de jogo. Esses 25 *clusters* são minerados de uma base de dados (BD) que contém 4000 estados de tabuleiros de

final de jogo (tabuleiros com 12 peças) obtidos de partidas jogadas entre os jogadores *VisionDraughts*, *NeuroDraughts* e *LS-Draughts*. Foi usada uma rede de *Kohonen-SOM* para minerar esses estados de tabuleiros [35], [36].

A dinâmica adotada pelo *MP-Draughts* durante as partidas de Damas é a seguinte: o *VisionDraughts* é o agente responsável por jogar desde o tabuleiro em seu estado inicial até que ele atinja a quantidade máxima de 12 peças. O jogador *VisionDraughts* corresponde ao agente *IIGA*. Do ponto de 12 peças em diante sobre o tabuleiro, um dos 25 agente especializados em final de jogo assume o mesmo. É escolhido, por meio de Redes *Kohonen-SOM*, qual dos agentes de final de jogo (entre os 25 treinados para final de jogo) melhor representa o estado de tabuleiro de final de jogo corrente. O modo como essa escolha é feita será apresentado na seção 5.4.4.

As próximas seções abordam, em detalhe, o desenvolvimento do jogador *MP-Draughts*. Inicialmente será apresentada a ordenação feita na árvore de busca do jogador *VisionDraughts*. Faz-se necessária a explicação dessa ordenação, pois como dito no capítulo 4, o *VisionDraughts* é a base sobre a qual se fundamenta o presente jogador e sua versão original não possuía tal ordenação. Como a ordenação da árvore aconteceu em paralelo com o desenvolvimento do *MP-Draughts*, achou-se conveniente que sua explicação fosse tratada neste capítulo. Na sequência, serão apresentados: a arquitetura do jogador multi-agente; os agentes que compõem tal arquitetura; a ferramenta utilizada na implementação do jogador; assim como os resultados obtidos pelo mesmo.

5.1 Ordenação da árvore de busca do *VisionDraughts*

O módulo de busca eficiente, em árvores de jogos, foi desenvolvido no *VisionDraughts* para fornecer ao jogador automático maior capacidade de analisar jogadas futuras, ou seja, maior capacidade de analisar estados do tabuleiro mais distantes do estado corrente. A introdução de tal módulo de busca proporcionou ao jogador acesso a jogadas já analisadas em situações anteriores [11]. Essas jogadas são armazenadas em uma tabela de transposição e são elas que possibilitam a ordenação da árvore de busca.

Como abordado na seção 4.6.2, as tabelas de transposição são utilizadas em conjunto com aprofundamento iterativo para produzir árvores de busca parcialmente ordenadas. Entre as informações armazenadas em uma tabela de transposição para um determinado estado, está a melhor ação a ser executada a partir dele. Quando o aprofundamento iterativo pesquisar um nível mais profundo da árvore de busca e revisitar um dado estado S_0, o filho S_m de S_0 originado pela execução da melhor ação, eventualmente, indicada na tabela de transposição para ser executada a partir de S_0 será pesquisado primeiro, desde que a árvore esteja ordenada. Assumindo que uma busca mais rasa é uma boa

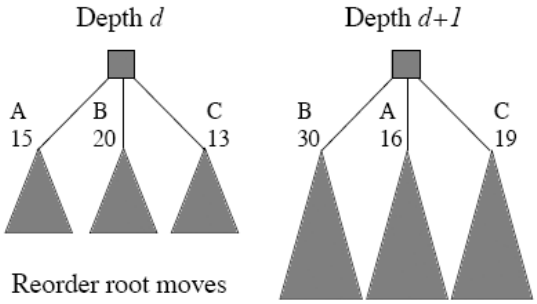

Figura 5.1: Exemplo de ordenação da árvore de busca com aprofundamento iterativo

aproximação para outra mais profunda, a melhor ação para um estado S_0 na profundidade d será, possivelmente, a melhor ação para o estado S_0 na profundidade $d + 1$ [56]; e com a ordenação da árvore colocando o melhor movimento no primeiro ramo da árvore de busca (ramos mais a esquerda), o melhor movimento deverá ser encontrado mais rapidamente.

A melhor ação citada acima, na verdade, é o filho do estado(encontrado na tabela) que obteve melhor predição durante as buscas feitas anteriormente. A ordenação parcial dos nós consiste em fazer com que esse filho ocupe a primeira posição da esquerda para a direita da árvore de busca. Logo, a ordenação faz com que o melhor movimento, a partir de um estado na profundidade *d*, seja o primeiro a ser explorado na profundidade *d+1*.

No aprofundamento iterativo (usado em conjunto com a tabela de transposição), a busca começa pesquisando com profundidade igual a um, sendo, posteriormente, incrementada passo a passo. Em razão do crescimento exponencial da árvore, o tempo necessário para expandir seu nível mais profundo é muito superior ao tempo necessário para cada um dos níveis mais rasos. Os benefícios mais evidentes do uso do aprofundamento iterativo para os programas jogadores são: obtenção de um mecanismo eficiente de controle de tempo e obtenção de árvores de buscas parcialmente ordenadas [55], [56].

No *MP-Draughts* , a ordenação parcial é conseguida ordenando o melhor movimento (melhor filho) do estado pai, jogando-o para o ramo mais à esquerda da árvore, como mostrado na figura 5.1. Veja que a iteração d retornou o filho B como melhor movimento a ser realizado a partir da raiz ($20 > 15 > 13$). Assumindo que o resultado obtido pela iteração d contenha uma boa aproximação do melhor movimento a ser realizado na iteração *d+1*, a árvore de busca é ordenada de forma que o filho B fique mais à esquerda da mesma. No exemplo, após a execução da iteração *d+1*, o filho B mostrou-se, realmente, a melhor opção de movimento, com uma predição igual a 30.

O pseudo-código abaixo apresenta o eficiente módulo de busca do *VisionDraughts* acrescido da ordenação parcial da árvore de busca. Abaixo são explicados os pontos essenciais para o entendimento do processo de ordenação, sendo que o funcionamento do módulo de busca já foi apresentado na seção 4.6.3 desta dissertação.

Pseudo-código do algoritmo *fail-soft* alpha-beta com tabela de transposição, bases de dados de finais de jogos e ordenação

```
1: alfaBeta(node:n, int:depth, int:min, int:max, move:bestmove)
2: if leaf(n) or depth=0 then
3:    besteval := evaluate(n)
4:    store(n, besteval,bestmove,depth,hashExact)
5:    return besteval
6: end if
7: if n is a max node then
8:    besteval := min
9:    if !leaf(n) and !depth=0 then
10:      found= retrieve(child,besteval,bestmove,depth-2,parentIsMaxNode)
11:      if if (found != 0 AND scoreType!= hINVALID) then
12:         setChildrenOrder(n,bestmove)
13:      end if
14:   end if
15:   for each child of n do
16:      if retrieve(child,besteval,bestmove,depth-1,parentIsMaxNode) then
17:         v := besteval
18:      else
19:         v := alfabeta(child,depth-1,besteval,max,bestmove)
20:      end if
21:      if v > besteval then
22:         besteval:= v
23:         thebest = bestmove
24:      end if
25:      if besteval >= max then
26:         store(child,besteval,bestmove,depth,hashAtLeast)
27:         return besteval
28:      end if
29:   end for
30: end if
31: if n is a min node then
32:   besteval := max
33:   if !leaf(n) and !depth=0 then
34:      found= retrieve(child,besteval,bestmove,depth-2,parentIsMinNode)
35:      if if (found != 0 AND scoreType!= hINVALID) then
36:         setChildrenOrder(n,bestmove)
37:      end if
38:   end if
39:   for each child of n do
```

```
40:       if retrieve(child,besteval,bestmove,depth-1,parentIsMinNode) then
41:         v := besteval
42:       else
43:         v := alfabeta(child,depth-1,besteval,max,bestmove)
44:       end if
45:       if v < besteval then
46:         besteval:= v
47:         thebest = bestmove
48:       end if
49:       if besteval <= max then
50:         store(child,besteval,bestmove,depth,hashAtMost)
51:         return besteval
52:       end if
53:     end for
54: end if
55: bestmove = thebest
56: store(n,besteval,bestmove,depth,hashExact)
57: return besteval
```

- **linha 1**: o algoritmo alfa-beta recebe um estado do tabuleiro do jogo de Damas, *n*, uma profundidade de busca, *d*, e um parâmetro de saída para armazenar a melhor ação a ser executada, *bestmove*, a partir de *n*, um intervalo de busca delimitado pelos parâmetros *min* (representando o limite inferior do intervalo de busca, alfa) e *max* (representando o limite superior do intervalo de busca, beta). O retorno do algoritmo é a predição associada ao estado *n* na forma de um número real que corresponde à avaliação do estado *n* do ponto de vista do agente jogador;
- **linha 2**: o algoritmo alfa beta escolhe a melhor jogada a ser executada (armazena a melhor jogada no parâmetro de saída *bestmove*) fazendo uma busca em profundidade, da esquerda para a direita, na árvore do jogo. Por se tratar de um procedimento recursivo, exige-se uma condição de parada. A condição de parada do algoritmo verifica se o estado do tabuleiro *n* é uma folha da árvore;
- **linha 3**: o retorno da função *evaluate(n)* (que indica a predição dada pela rede neural para o estado do tabuleiro *n*) é armazenada em *besteval*;
- **linha 4**: quando o algoritmo de busca chamar a rede neural na linha 3, com a rotina *evaluate(n)*, para conhecer a predição associada ao estado folha *n*, o valor da predição recém-calculada deverá ser armazenado na tabela de transposição. Assim, o algoritmo invoca o procedimento *store*. Como *n* é um nó folha, o valor da predição deve ser armazenado com *scoretype* = *hashExact*, ou seja, predição exatamente igual a *besteval* ;

- **linha 7**: verifica se o estado do tabuleiro presente na raiz da árvore do jogo montada pelo algoritmo alfa-beta é um nó maximizador;
- **linha 10**: se o tabuleiro *n* não for uma folha da árvore de busca e não for a raiz da árvore, busca-se na tabela de transposição o melhor movimento (*bestmove*) armazenado para a profundidade *d-2*. O valor buscado é armazenado na variável *found*;
- **linha 11**: verifica se o valor de *found* é diferente de 0, o que indica que há um *bestmove* associado ao tabuleiro *n* para a profundidade *d-2* e que este valor realmente corresponde a uma predição *scoreType* válida. Essas são condições necessárias para se poder usar o valor encontrado na tabela;
- **linha 12**: o procedimento *setChildrenOrden(n, bestmove)* é invocado. Após a execução desse procedimento o ramo esquerdo da árvore de busca conterá o melhor movimento obtido para a profundidade *d-2*;
- **linha 15**: para cada um dos filhos de um estado maximizador *n*, antes de chamar a rotina de busca recursivamente, o repositório de dados em memória (tabela de transposição) deve ser consultado. Então o algoritmo invoca o procedimento *retrieve*, com *nodeType* = *parentIsMaxNode*. Outro detalhe muito importante é que, nesta linha, a profundidade de busca deve ser igual a *depth - 1* que corresponde à profundidade de *n*;
- **linha 16**: Se o filho do estado *n* estiver armazenado na tabela de transposição e o método *retrieve* obtiver sucesso no tratamento das informações, o valor da predição retornado por *retrieve* deve ser utilizado, ao invés de chamar o algoritmo alfa-beta recursivamente;
- **linha 25**: detectada a ocorrência de uma poda *beta*, quando o algoritmo estiver analisando os filhos de um nó *n* do tipo maximizador, significa que os demais filhos de *n*, caso existam, não precisam mais ser analisados. A variável *besteval* conterá o valor mínimo aceitável para o estado do tabuleiro representado pelo nó *n*. Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store*, com o *flag hashAtLeast* indicando que a predição associada ao estado do tabuleiro *n* é, no mínimo, igual a *besteval*;
- **linhas de 31 a 54**: são realizados os mesmos procedimentos das linhas 7 à 30, porém considerando o nó da raiz um nó minimizador;
- **linha 55**: quando o algoritmo tiver analisado todos os filhos do nó *n*, a variável *besteval* conterá exatamente o valor da predição para o estado do tabuleiro representado pelo nó *n*. Assim, a predição presente na variável *besteval* deve ser armazenada na tabela de transposição pelo método *store*, com o *flag hashExact* indicando que a predição associada ao estado do tabuleiro *n* é exatamente igual a *besteval*.

A cada nível da árvore de busca, o movimento com maior probabilidade de ser o melhor ocupará sempre a primeira posição (da esquerda para a direita), sendo o primeiro a ser explorado pelo processo de busca. Isso faz com que o processo de busca fique mais ágil, e que as chances de podas dos irmãos desse nó (melhor movimento) aumentem. Uma vez que torna o processo de busca mais eficiente (graças ao aumento do número de podas), o aprofundamento iterativo consegue explorar níveis mais profundos da árvore na tentativa de encontrar melhores movimentos, aumentando a visão (*look-ahead*) do jogador. Jogador com um campo de visão maior tem maiores condições de fazer escolhas mais acertadas e, consequentemente, melhorar sua eficiência.

O principal motivo da aceleração do processo de busca está diretamente ligado ao fato do aumento do número de podas. A constatação desse aumento significa que os melhores movimentos realmente estão sendo encontrados nos primeiros nós da árvore de busca, confirmando a eficiência da ordenação e o que Plaat aponta em [55].

Para ilustrar a melhora do jogador *VisionDraughts* incrementado com o módulo de ordenação da árvore de busca, o mesmo foi comparado com o *VisionDraughts* sem o módulo de ordenação, o que expõe claramente tal contribuição. Foi realizado um torneio de 14 jogos em que o jogador com o módulo de ordenação venceu 8 partidas, empatou 5 e perdeu somente uma. Quanto aos casos de *loops* de final de jogo encontrados durante o treinamento do *VisionDraughts*, os seguintes resultados puderam ser observados: em um treinamento de 2000 jogos, no jogador sem a ordenação ocorreram 96 casos de *loop*, enquanto que no jogador com a ordenação, a ocorrência de *loop* foi de 41 casos. Os casos de *loop* diminuíram em 52% no jogador com o módulo de ordenação

Portanto, a ordenação da árvore de busca contribuiu para diminuir, consideravelmente, o número de partidas encerradas indevidamente em *loop* durante o treinamento, o que impactou, diretamente, na eficiência do jogador.

5.2 Arquitetura Geral do *MP-Draughts*

Conforme explicado anteriormente, *MP-Draughts* é um sistema multiagente para jogos de Damas. Cada agente que compõe o jogador consiste de uma rede neural multicamadas treinada pelo método de aprendizagem por reforço TD(λ). Esse método utiliza predições sucessivas para ensinar a rede neural a jogar Damas. O agente alcança alto nível de jogo simplesmente jogando contra si próprio (estratégia de treino por *self-play* com clonagem), com mínima busca e sem qualquer análise de jogos feitas por especialistas humanos.

A arquitetura do *MP-Draughts* é mostrada na figura 5.2 e resumida logo a seguir:

1. ***IIGA*** - *Initial/Intermidiate Game Agent*: esse agente é treinado para ser especialista em fases iniciais e intermediárias de jogo, o que aqui representa responsável por atuar em tabuleiros com, no mínimo, 13 peças. O agente *IIGA* nada mais é que

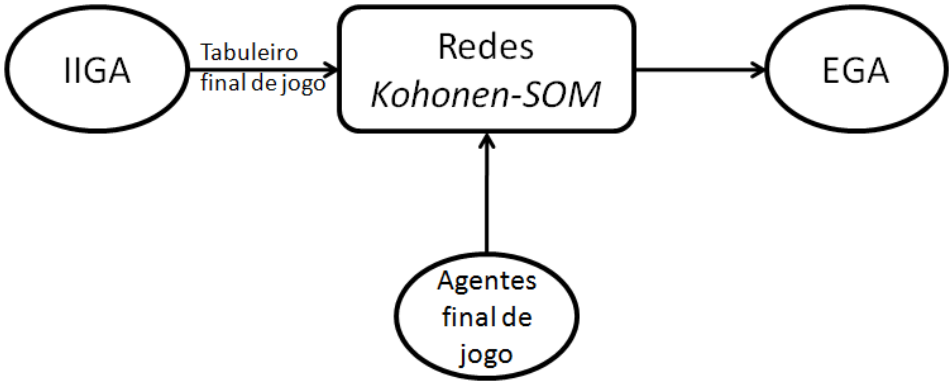

Figura 5.2: Arquitetura geral do *MP-Draughts*: um sistema multiagente de aprendizagem de jogos de Damas.

a rede jogadora de Caixeta e Julia [11], o *VisionDraughts*, acrescida do processo de ordenação na árvore de busca;

2. ***Agentes de Final de Jogo***: esse módulo consiste de 25 redes jogadoras similares ao agente jogador *IIGA*. Todavia elas foram treinadas a partir de um conjunto de tabuleiros de final de jogo (tabuleiros com 12 peças) distribuídos em 25 *clusters* que foram organizados por meio de redes *Kohonen-SOM* [35], [34], [36]. Os estados que compõem tais *clusters* são provenientes de uma base de dados composta de 4000 estados de tabuleiro de final de jogo. A cada *cluster* foi acoplada uma rede neural similar à rede usada no IIGA. Ao final do treinamento, cada rede é capaz de lidar com um determinado tipo de estado de tabuleiro de final de jogo diferente (definido pelo *cluster* que a rede representa);
3. **Rede *Kohonen-SOM***: o módulo de Rede *Kohonen-SOM* também é responsável por classificar, entre os 25 agentes de final de jogo, qual está mais apto a assumir a fase de final de jogo. Esse módulo tem como entrada o tabuleiro que representa o estado corrente do jogo (tabuleiro com no máximo 12 peças) e um vetor de pesos que representa cada agente de final de jogo. Dado o vetor que representa o estado atual do jogo e os vetores de pesos referentes aos agentes de final de jogo, a rede *Kohonen-SOM* calcula a distância entre o vetor que representa o estado corrente do jogo e todos 25 vetores que representam os agentes. Feito esse cálculo, o vetor de pesos que tiver menor distância do vetor do estado corrente será selecionado. A rede que estiver associada a este vetor de peso será acionada de modo a finalizar o jogo corrente. Ela também corresponde ao *EGA* definido a seguir;
4. ***EGA*** - *EndGame Agent*: o *EGA* é o agente de final de jogo escolhido, entre os 25 agentes de final de jogo existentes, para finalizar o jogo de Damas. Esse agente, por ser o mais próximo do estado atual do tabuleiro, é o que tem maior condições de finalizar a partida, visto que o mesmo foi treinado em estados de tabuleiros similares ao estado alcançado pelo *IIGA* durante o jogo.

Como abordado anteriormente, cada agente que compõe o *MP-Draughts* apresenta a mesma arquitetura do jogador *VisionDraughts* (apresentada na seção 4.2), acrescida da ordenação da árvore de busca, ou seja, uma rede neural MLP que usa Diferenças Temporais e algoritmo de busca alfa-beta para escolher o melhor movimento, além disso, usa estratégia de treinamento por *self-play* com clonagem e com o mínimo de intervenção humana possível.

As seções seguintes apresentam, em detalhes, a estrutura de cada agente que compõe o *MP-Draughts* e os resultados obtidos por ele.

5.3 *IIGA - Initial/Intermidiate Game Agent*

O jogador automático de Damas *VisionDraughts* [11] é a base sobre a qual se fundamenta o jogador *IIGA*.

Apesar de o *IIGA* ter a mesma estrutura do *VisionDraughts*, algumas adaptações foram feitas no *VisionDraughts* para que ele atendesse as necessidades do *MP-Draughts*. Essas necessidades são: diminuir o máximo possível a interferência humana no processo de aprendizagem do jogador; e adaptar o conjunto de características usada pelo jogador de modo a inserir características essenciais tanto para começo quanto para final de jogo.

Para reduzir a interferência humana no processo de aprendizagem do jogador, a possibilidade de utilização da base de dados com jogadas de final de jogo admitida pelo *VisionDraughts* foi retirada da estrutura do *IIGA*. Com a retirada dessa bas,e o jogador não terá mais "suporte"para saber se um dado estado S de tabuleiro (com oito peças ou menos) representa estado de vitória, empate ou derrota antes que o jogo termine. Para saber o resultado do jogo envolvendo o estado S de tabuleiro, o jogador terá que conduzi-lo até o final.

O módulo de acesso à base de dados de final de jogo fornece ao agente jogador capacidade de anunciar, antes do final da partida, se um estado do tabuleiro com até oito peças representa vitória, derrota ou empate. A utilização da base de dados para as fases finais do jogo agiliza o processo de escolha da melhor ação a ser executada por um jogador automático. Além disso, torna mais eficiente o processo de escolha da melhor ação na medida em que substitui heurística por informação perfeita presente nas bases (vitória, derrota ou empate). Como a base de dados foi retirada no *IIGA*, o processo de treinamento ficou mais demorado, e as escolhas de movimentos já não são tão precisas como quando é utilizada a base de dados. A eficiência do jogador *IIGA* foi prejudicada pela retirada da base de dados, mas foi favorecida quando se observa o fator "interferência humana", visto que esse fator foi consideravelmente reduzido, o que atende a expectativa em relação ao jogador proposto.

No que tange às características de jogo de Damas usadas pelo jogador durante o processo de aprendizagem, a seleção delas continua sendo feita, manualmente, assim como é

feita no *VisionDraughts* e no *NeuroDraughts*. Todavia, as características usadas no *IIGA* contêm duas características a mais do que as usadas pelo *VisionDraughts*: *diagonalmoment* e *threat*, totalizando 14 características (conforme tabela 5.1), ou seja, a representação NET-FEATUREMAP do tabuleiro é obtida pelo mapeamento C: B $\rightarrow \mathbb{N}^{14}$.

Cada atributo do tabuleiro NET-FEATUREMAP (ver seção 4.1.2) é convertido para sua representação binária respeitando-se a quantidade de *bits* previstos para as características adotadas (segunda coluna da tabela 5.1). Cada um desses *bits* alimentará um neurônio da entrada da rede MLP.

CARACTERÍSTICAS	N° BITS
PieceAdvantage	4
PieceDisadvantage	4
PieceThreat	3
PieceTake	3
Advancement	3
DoubleDiagonal	4
BackRowBridge	1
CentreControl	3
XCentreControl	3
TotalMobility - MOB	4
Exposure	3
KingCentreControl	3
Threat	3
DiagonalMoment	4

Tabela 5.1: Conjunto de Características implementadas no jogador *MP-Draughts*

A figura 5.3 mostra o novo fluxo de aprendizagem do jogador sem a base de dados de final de jogo. Veja que os módulos são bastante parecidos com os do sistema *VisionDraughts*. A exceção é a base de dados de final de jogo que não está presente na estrutura do *IIGA*.

O fluxo mostrado na figura 5.3 é resumido a seguir:

1. **Estado Corrente** → percepção → **Módulo Eficiente de Busca**: o algoritmo de busca eficiente recebe como parâmetro de entrada o estado corrente do tabuleiro do jogo. A partir de então, ele monta uma árvore de busca com intuito de descobrir qual a melhor jogada a ser executada;
2. **Módulo Eficiente de Busca** → nós folhas → **Módulo NET-FEATUREMAP**: os estados da camada mais profunda da árvore de busca, chamados folhas, são enviados para o módulo NET-FEATUREMAP, que aplicará a conversão de tabuleiro aos mesmos. O mapeamento NET-FEATUREMAP é feito com base em funções que descrevem as próprias características do jogo de Damas [42], [43];

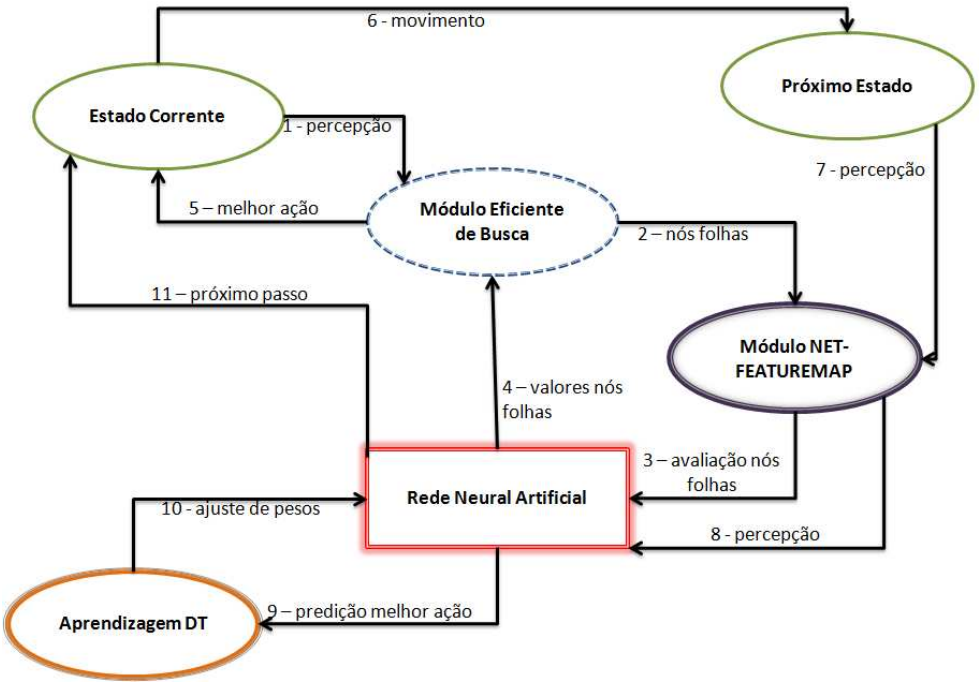

Figura 5.3: Fluxo de aprendizado do *IIGA*: um sistema de aprendizagem para fases iníciais/intermediárias de jogos de Damas

3. **Módulo NET-FEATUREMAP** → avaliação nós folhas → **Rede Neural Artificial**: o módulo NET-FEATUREMAP converte as folhas da árvore mapeando-as na camada de entrada da rede neural multicamadas;

4. **Rede Neural Artificial** → valores nós folhas → **Módulo Eficiente de Busca**: a rede neural multicamadas recebe um estado do tabuleiro mapeado em sua camada de entrada e retorna, em seu único neurônio da camada de saída, um valor entre -1.0 e +1.0, representando a avaliação do estado de entrada sob a ótica do jogador automático. Tal valor, denominado predição do estado de entrada, é retornado para o módulo de busca que o utilizará com o propósito de descobrir a melhor ação a ser executada;

5. **Módulo Eficiente de Busca**→ melhor ação → **Estado Corrente**: após montar a árvore de busca e utilizar os módulos de mapeamento e rede neural para avaliar as folhas da árvore, o algoritmo de busca propaga, de baixo para cima, a melhor ação a ser executada pelo agente jogador;

6. **Estado Corrente** → movimento → **Próximo Estado**: de posse da melhor ação a ser executada, o estado do tabuleiro é modificado com a realização de uma ação concreta para um próximo estado;

7. **Próximo Estado**→ percepção → **Módulo NET-FEATUREMAP**: esse novo estado do tabuleiro do jogo é enviado para o módulo NET-FEATUREMAP , como anteriormente.

8. **Módulo NET-FEATUREMAP** → percepção → **Rede Neural Artificial**: o

módulo NET-FEATUREMAP , agora, mapeia esse novo estado diretamente na entrada da rede neural multicamadas. Note que agora, não se usa o módulo de busca;

9. **Rede Neural Artificial** → predição melhor ação → **Aprendizagem TD**: assim que o novo estado é mapeado na camada de entrada da rede neural, um novo valor (predição) é obtido no seu único neurônio de saída;
10. **Aprendizagem DT** → ajuste de pesos → **Rede Neural Artificial**: essa nova predição, recém calculada, é utilizada juntamente com a última predição, anteriormente calculada no neurônio de saída, para atualizar todos os pesos da rede neural multicamadas;
11. **Rede Neural Artificial**: → próximo passo → **Novo Estado Tabuleiro** : a partir de agora, com os pesos da rede atualizados, o jogo passa para um novo estado de tabuleiro (estado atingido depois de um movimento);
12. **Novo Estado Tabuleiro**: → próximo passo → **Estado Corrente**: o fluxo retorna ao estágio inicial e o procedimento começa a se repetir até o fim de uma partida de treinamento.

O processo de treinamento do *IIGA* é o mesmo seguido pelo *VisionDraughts* [11]: *self-play* com clonagem. A ideia básica do treinamento por *self-play* com clonagem é treinar um jogador em vários jogos contra uma cópia de si próprio. A partir de um certo momento, o oponente com menor nível de desempenho é descartado, e o oponente com maior nível de desempenho é clonado para que outros jogos sejam realizados, e o novo jogador com o maior nível de desempenho seja selecionado para clonagem. O processo se repete até que um jogador com alto nível de desempenho seja obtido.

5.4 *EGA - EndGame Agent*

O *MP-Draughts* é composto por 26 agentes, sendo um agente (***IIGA***) especialista em início e meio de jogo (descrito na seção anterior) e outros 25 agentes especialistas para final de jogo. Todos os 25 agentes especializados em final de jogo possuem a mesma arquitetura do *IIGA*, porém treinados em estados de tabuleiro de final de jogo.

Cada um dos 25 agentes será treinado em diferentes estados de tabuleiro de final de jogo, sendo que cada agente será treinando e especializado para lidar com um determinado tipo de tabuleiro. Os tabuleiros usados para treinar os agentes de final de jogo são agrupados em 25 *clusters* diferentes, em que cada *cluster* corresponderá, no final do processo de treinamento, a um agente especializado em um determinado tipo de tabuleiro de final de jogo. Cada *cluster* usado para treinar os agentes de final de jogo contém 25 estados de tabuleiros que são minerados (por meio de redes de *Kohonen-SOM*) de uma

base de dados com 4000 estados de tabuleiro. Os estados de tabuleiro pertencentes a um dado *cluster*, apesar de distintos entre si, satisfazem um critério mínimo de similaridade, fato que não ocorre entre esse *cluster* e os demais *clusters*.

No caso, o critério de similaridade usado para agrupar os estados de tabuleiro nos *clusters* é baseado nas caracteríticas, conforme apresentado na seção 5.4.2.

O *MP-Draughts* usa uma extensão do *VisionDraughts* como seu jogador de início de partida (*IIGA*); por isso ele foi usado para gerar a base de dados com tabuleiros de final de jogo usada no treinamento dos agentes de final de jogo.

Nas subseções que seguem, serão apresentados os passos seguidos para obter agentes especializados em final de jogo, que é o foco principal deste trabalho. A apresentação varrerá desde a geração da base de dados usada para treinar o jogador, até a obtenção do jogador especializado em final de jogo.

5.4.1 Obtenção de Base de Dados de Treinamento dos Agentes de Final de Jogos

Para o treinamento dos agentes de final de jogo, foi gerada uma base de dados contendo 4000 estados de tabuleiros de final de jogo, ou seja, tabuleiros com 12 peças.

A escolha de usar tabuleiros com 12 peças para representar final de jogo foi feita após a observação de vários estágios de um jogo de Damas. No estágio de jogo com 12 peças sobre o tabuleiro, ainda não existe um jogador tendencioso a ganhar o jogo, ao contrário do que pôde ser observado em tabuleiros com 10 ou menos peças. Em tabuleiros com 10 peças ou menos, a vantagem de um jogador em relação ao seu oponente, na maioria dos casos, já é visível e a possibilidade de reversão de resultado de jogo fica comprometida, o que não é interessante para os agentes de final de jogo. Desse modo, fica definido como fase de final de jogo, tabuleiros com 12 peças.

Em se tratando de tabuleiros com 12 peças, nota-se que existe cerca de 1 quatrilhão de configurações possíveis de tabuleiros [71], [69], como pode ser observado na figura 5.4. Assim sendo, mesmo considerando os mais modernos recursos computacionais, é impraticável treinar agentes para serem especialistas em todos esses estados.

Como o número de configurações possíveis de tabuleiros com 12 peças é muito grande, e entre essas configurações existem tabuleiros cuja ocorrência é praticamente impossível de acontecer durante um jogo (tabuleiros com 12 peças rainhas, por exemplo), foram realizados torneios entre alguns jogadores automáticos de Damas na intenção de verificar quais estados de tabuleiros com 12 peças são mais prováveis. Tais torneios envolveram os jogadores *NeuroDraughts* [42], *LS-Draughts* [50], VisionDraughts [11] e o *IIGA*. Observou-se que os estados alcançados por esses jogadores são realistas e tem uma boa chance de ocorrerem durante uma partida de jogo de Damas, isso porque esses estados foram obtidos durante partidas reais de jogo de Damas.

Pieces	Number of positions
1	120
2	6,972
3	261,224
4	7,092,774
5	148,688,232
6	2,503,611,964
7	34,779,531,480
8	406,309,208,481
9	4,048,627,642,976
10	34,778,882,769,216
Total 1–10	39,271,258,813,439
11	259,669,578,902,016
12	1,695,618,078,654,976
13	9,726,900,031,328,256
14	49,134,911,067,979,776
15	218,511,510,918,189,056
16	852,888,183,557,922,816
17	2,905,162,728,973,680,640
18	8,568,043,414,939,516,928
19	21,661,954,506,100,113,408
20	46,352,957,062,510,379,008
21	82,459,728,874,435,248,128
22	118,435,747,136,817,856,512
23	129,406,908,049,181,900,800
24	90,072,726,844,888,186,880
Total 1–24	500,995,484,682,338,672,639

Figura 5.4: Espaço de estados para o jogo de Damas: quantidade de estados possíveis de acordo com o número de peças sobre o tabuleiro (SCHAEFFER et al., 2007).

O torneio entre os jogadores foi distribuído da seguinte maneira:

- 1000 jogos entre *VisionDraughts* e *NeuroDraughts*;
- 1000 jogos entre *VisionDraughts* e *LS-Draughts*;
- 1000 jogos entre *VisionDraughts* e *IIGA*;
- 1000 jogos entre *LS-Draughts* e *NeuroDraughts*.

Inicialmente foi gerada uma base de dados contendo 1000 estados de tabuleiros de final de jogo obtidos de jogos entre esses jogadores. Porém essa quantidade não foi suficiente para obter uma boa distribuição dos estados entre os *clusters*, de modo que alguns clusters se encontravam vazios ao final do processo de *clusterização*. Foram feitos também testes com bases de 2000 e 3000 estados, porém os melhores resultados foram obtidos com 4000 estados.

Após o término do torneio, a base de dados com 4000 estados de tabuleiros já está montada. No intuito de selecionar em quais desses tabuleiros cada agente de final de jogo será treinado, deve-se aplicar uma técnica de *clusterização* sobre a base para obter os grupos de treinamentos. A técnica de *clusterização* usada foi redes *Kohonen-SOM* [36], [38] que retorna, como produto final de sua execução, grupos de dados. Os dados pertencentes a um determinado grupo apresentam similaridades satisfatórias entre si, diferentemente, dos pertencentes aos outros grupos. O processo de *clusterização* bem como a regra de similaridade usada para agrupar os estados serão apresentados na seção 5.4.2 deste capítulo.

5.4.2 *Clusterização* da Base de Dados usando Redes de *Kohonen-SOM*

As redes *Kohonen-SOM* é um modelo de rede neural artificial desenvolvido por Teuvo Kohonen [35], [34] para demonstrar as possibilidades de uma rede competitiva trabalhar de forma não-supervisionada. Essas redes neurais são compostas por duas camadas, sendo a primeira, de entrada e a segunda, de saída. A primeira camada é por onde os dados a serem classificados são apresentados aos nodos que compõem a camada de entrada. Os nodos dessa camada são completamente interconectados com os neurônios da camada de saída, a qual é competitiva. Na camada de saída ocorrem as competições entre os neurônios. A competição se dá por meio do cálculo da menor distância entre os nodos da camada de entrada e cada neurônio da camada de saída. Terminada cada competição, o neurônio vencedor e seus dois vizinhos (um à esquerda e outro à direita) têm seus pesos atualizado conforme a entrada dada.

A utilização de uma rede de *Kohonen-SOM* é dada em duas etapas: uma de treinamento, em que os pesos são ajustados conforme um determinado conjunto de treinamento; e uma etapa de utilização, em que os pesos não são ajustados e os *clusters* são formados.

As redes de Kohonen-SOM são treinadas para aprender a classificar dados em grupo segundo suas similaridades. Tal aprendizado insere-se no contexto dos aprendizados não-supervisionados [60], ou seja, o reajuste dos pesos das redes são atualizados sem interferência externa (fato que exclui os reforços e as comparações com padrões esperados).

Nesse ponto, já é de conhecimento do leitor o processo de formação da base de dados de final de jogo, a representação dos tabuleiros na base de dados e como eles devem ser apresentados à rede *Kohonen-SOM*. Sendo assim, já é possível entender como funciona o processo de *clusterização* dos estados de final de jogo presentes na base de dados, apresentados na seção seguinte.

Arquitetura Geral do Processo de *Clusterização*

A arquitetura geral do processo de *clusterização* da base de dados do jogador *MP-Draughts* é apresentada na figura 5.5.

Os módulos que compõem essa arquitetura são:

- ***Base de dados*:** composta por 4000 estados de tabuleiros de final jogo (tabuleiros com 12 peças) obtidos de torneios entre os jogadores automáticos de Damas *VisionDraughts*, *NeuroDraughts*, *LS-Draughts* e *IIGA*. Essa base contém os estados de tabuleiros que serão usados no processo de treinamento dos agentes especialistas em final de jogo. Os tabuleiros presentes nessa base são representados na forma vetorial (descrito na seção 4.1.1), visto que foram obtidos de partidas envolvendo jogadores que utilizam dessa representação durante os jogos. Como mostrado anteriormente,

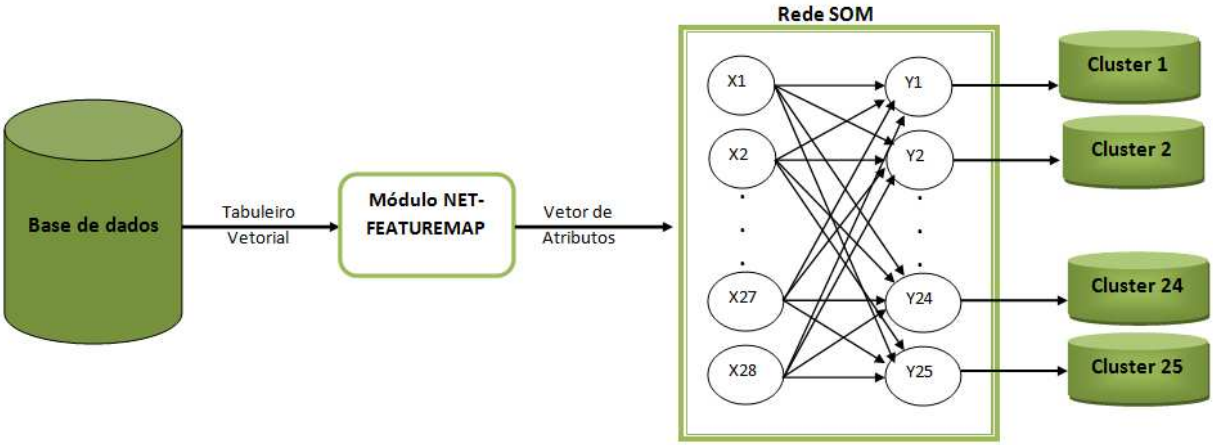

Figura 5.5: Arquitetura do processo de *Clusterização*

a representação vetorial é a mais usada pelos jogadores automáticos de Damas por ser uma forma de representação simples e eficiente;

- ***Módulo NET-FEATUREMAP*:** este módulo é o responsável por converter um tabuleiro de representação vetorial em um vetor de atributos (representação por características). Nesse caso, os estados pertencentes à base, inicialmente, representados de modo vetorial são convertidos para a representação NET-FEATUREMAP (por características), sendo que são usadas as 28 características de Samuel para representar o tabuleiro, ou seja, C: B $\rightarrow \mathbb{N}^{28}$ (seção 4.1.2). O módulo NET-FEATUREMAP recebe do módulo Base de Dados um tabuleiro vetorial, converte o tabuleiro em um vetor de atributo e o retorna para o módulo da Rede *Kohonen-SOM*. O vetor de atributos resultante do módulo NET-FEATUREMAP é a entrada do módulo Rede *Kohonen-SOM*.

- ***Rede Kohonen-SOM*:** esse módulo tem como entrada, o vetor de atributos gerado pelo módulo *NET-FEATUREMAP* e, como saída, 25 arquivos de tabuleiros. Cada elemento do vetor de atributos é conectado a um nodo da camada de entrada da rede *Kohonen-SOM*. Todos os nodos da camada de entrada são conectados a todos os neurônios da camada de saída (como mostrada na estrutura de rede *Kohonen-SOM*, explicado no capítulo 2). Quando um vetor é apresentado à entrada da rede *Kohonen-SOM* durante o treinamento, é calculado o quadrado da distância Euclidiana entre o estado apresentado à entrada da rede e cada neurônio de saída. O neurônio de saída que possuir menor distância em relação aos nodos da camada de entrada é o vencedor, devendo ser candidato a representar o tabuleiro de entrada. Nesse caso, a cada neurônio de saída, será associado um arquivo contendo a representação vetorial dos estados de tabuleiros que o neurônio representa.

  Detalhes referentes ao cálculo da distância entre tabuleiros de entrada e neurônio de saída serão apresentados no algoritmo de *clusterização* desta seção.

- ***Clusters*:** O módulo *Clusters* nada mais é que bases de dados (com estados de tabuleiros de final de jogo) criadas a partir dos arquivos de saída do módulo Rede *Kohonen-SOM* . Cada um dos 25 *clusters*, presentes nesse módulo, representa um neurônio de saída da rede *Kohonen-SOM*. Um estado pertencente a um dado *cluster* tem alto grau de similaridade em relação aos outros estados desse *cluster*, ou seja, possui várias características de jogo de Damas em comum. Ao final da execução do processo de *clusterização*, cada *cluster* deve conter 25 estados de tabuleiro os quais serão usados para treinar um agente de final de jogo específico. A redução do número de estados presentes em um *cluster* foi feita no intuito de diminuir o tempo de treinamento do agente de final de jogo, visto que tais agentes demandam muito tempo de treinamento.

O fluxo de dados que acontece durante o processo de *clusterização*, apresentado na figura 5.5, é resumido a seguir:

- **Base de dados** → tabuleiro vetorial → **Módulo NET-FEATUREMAP**: a base de dados passa, como parâmetro de entrada para o módulo *NET-FEATUREMAP*, os estados de tabuleiro de final de jogo na forma de representação vetorial. É passado um estado de tabuleiro de cada vez. A partir de então o módulo *NET-FEATUREMAP* converte o tabuleiro de representação vetorial em um vetor de atributos (representação por características);
- **Módulo NET-FEATUREMAP** → vetor de atributos → **Rede *Kohonen-SOM***: depois que o tabuleiro vetorial foi convertido em vetor de atributos, ele é passado como parâmetro de entrada para a rede neural. A rede neural recebe o vetor de atributos, conecta cada posição do vetor a um nodo específico da camada de entrada, verifica qual neurônio de saída mais se assemelha com vetor de atributos e elege-o como neurônio "vencedor". Escolhido o neurônio vencedor, ele recebe o tabuleiro vetorial correspondente ao vetor característica conectado na camada de entrada da rede;
- **Rede *Kohonen-SOM*** → tabuleiro vetorial → **Clusters**: depois que o neurônio de saída foi escolhido, o tabuleiro vetorial correspondente ao vetor característica conectado na entrada da rede neural é direcionado para o *cluster* que representa o neurônio vencedor da camada de saída. Esse processo se repete até que todos os estados da base tenham sido *clusterizados*. No caso específico dessa base de dados, os passos apresentados aqui se repetem por 4000 vezes.

O processo de *clusterização* da base de dados é apenas um dos módulos que envolve o processo de aprendizagem dos agentes de final de jogo que compõem o *MP-Draughts*. Depois da fase de *clusterização*, vem a fase de treinamento dos agentes jogadores (redes neurais de final de jogo). A fase de treinamento das redes de final de jogo é análoga

ao do *IIGA*, diferindo apenas, na quantidade e na natureza dos estados de tabuleiros a partir dos quais as redes são treinadas. Enquanto o *IIGA* é treinado a partir de um único tabuleiro (o de início de jogo de Damas), cada agente de final de jogo é treinado a partir de 25 estados de tabuleiro de final de jogo.

Ainda no processo de entendimento da fase de *clusterização*, a próxima seção traz a explicação do algoritmo usado pela rede *Kohonen-SOM* para *clusterizar* os estados da base.

O Algoritmo *Clusterização Kohonen-SOM*

O *MP-Draughts* utiliza o algoritmo *Kohonen-SOM* para agrupar os estados de tabuleiro de final de jogo presentes na base de dados. A medida de similaridade usada é o quadrado da distância Euclidiana.

Visando simplificar o entendimento do algoritmo de *clusterização Kohonen-SOM*, é apresentado, a seguir,o pseudo-código do algoritmo juntamente com uma descrição de cada linha.

Note que, das linhas 3 a 15, é feito o reajuste de peso na rede, ao passo que, das linhas 16 a 22, calcula-se qual neurônio de saída representará o estado de final de jogo analisado.

Pseudo-código do algoritmo de *clusterização* por redes *Kohonen-SOM*

1: **Kohonen**(**int**:radius, **double**:alpha, **int**:numberNeuronsIn, **int**:numberNeuronsOut)
2: Initialize weights W_{ij}
3: **while** alpha > 0.1 **do**
4: **for** each input datum B_k: **do**
5: **for** each j, compute: **do**
6: D(j) = $\sum_{i=1}^{n} (W_{ij} - A_i)^2$
7: **end for**
8: Find Y_j such that $D(J)$ is a minimum
9: **for** all units j within a specified neighborhood of J, and for all i: **do**
10: W_{ij}(new) = W_{ij}(old) + α[X_i - W_{ij}(old)]
11: **end for**
12: **end for**
13: Update learning rate
14: Test stopping condition
15: **end while**
16: **for** each input datum B_k: **do**
17: **for** each j, compute: **do**
18: D(j) = $\sum_{i} (W_{ij} - A_i)^2$
19: **end for**
20: Find Y_j such that $D(J)$ is a minimum

21: Store B_k in Y_j

22: **end for**

Onde:

- **linha 1:** o algoritmo *Kohonen-SOM* recebe o raio que indica o número de vizinhos do neurônio de saída que está sendo avaliado cujos os vetores de pesos dos vizinhos serão reajustados juntamente com o neurônio de saída; a taxa de aprendizagem do algoritmo, o número de nodos da camada de entrada da rede (igual ao número de características implementadas) e o número de neurônios da camada de saída (igual ao número de *cluster* que devem ser gerados no final da execução do algoritmo). No caso específico deste algoritmo, o raio da vizinhança é 1, a taxa de aprendizagem começa em 1 e vai decrementando em 5% ao longo do treinamento, o número de nodos de entrada é 28 (correspondente ao número de características) e o número de neurônios de saída é 25 (número de *clusters* gerado no final do algoritmo). Ao final de execução do algoritmo, os vetores associados a cada neurônio devem ser armazenados em 25 arquivos diferentes, cada um representando um neurônio de saída em específico;
- **linha 2:** cada conexão W_{ij} no mapa de Kohonen, mostrado na figura 2.8 no capítulo 2, tem um peso associado. A rede é inicializada com valores aleatórios para esses pesos. Os valores de peso são atualizados durante o treinamento da rede. Cada nó vencedor atualiza seu próprio peso e, em menores graus, os nós da vizinhança também são atualizados;
- **linha 3:** a taxa de aprendizagem *alpha* é a condição de parada do algoritmo. Os pesos da rede são atualizados até que a taxa de aprendizagem seja menor que 0.1. Quando a taxa é atingida, considera-se que o algoritmo já convergiu e a rede está apta a classificar corretamente a base de dados;
- **linha 4:** o laço de execução é repetido até que todos os estados de tabuleiros B_k presentes na base de dados tenham sido analisados pela rede *Kohonen-SOM*. Para cada estado B_k, execute os passos de 5 a 12 do algoritmo;
- **linha 5:** dado um estado de tabuleiro, representado por características (C: B $\rightarrow$ $\mathbb{N}^{28}$) B_k, o quadrado da distância Euclidiana D_j entre esse tabuleiro e cada neurônio Y_j da camada de saída deve ser analisado, sendo que j corresponde ao neurônio de saída que está sendo analisado;
- **linha 6:** o cálculo do quadrado da distância Euclidiana $D_{(j)}$ entre o vetor de pesos que representa o neurônio de saída Y_j e o vetor de entrada da rede neural é calculado com base na fórmula: D(j) $=\sum_{i=1}^{n} (W_{ij} - A_i)^2$, onde n representa o número de características da representação NET-FEATUREMAP, i representa qual nó de entrada X_i está sendo analisado, o índice j especifica o neurônio de saída, W_{ij} é

valor do peso correspondente à conexão entre i e j, e A_i corresponde ao valor do atributo representado no nodo X_i da camada de entrada da rede neural. Como explicado anteriormente, cada posição do vetor característica é conectada a um nodo específico da camada de entrada, ou seja, um vetor de características com 28 posições é representado da seguinte maneira na entrada de rede *Kohonen-SOM*: $B_k = <A_1, A_2, ..., A_{27}, A_{28}>$. Repete-se esse processo até que todas as distâncias entre um dado vetor de entrada e todos os neurônios de saída sejam calculadas;

- **linha 8:** após calculadas todas as distâncias entre o vetor de entrada e os neurônios de saída Y_j, encontra-se o neurônio de saída Y_j com menor distância;
- **linhas 9 e 10:** o neurônio Y_j com menor distância tem seus pesos atualizados, assim como os seus vizinhos, sendo atualizado um vizinho à direita e um à esquerda desde neurônio. A atualização dos pesos é feita usando a seguinte fórmula: W_{ij}(new) = W_{ij}(old) + α[X_i - W_{ij}(old)], em que W_{ij}(old) é o peso antigo do neurônio. Após o ajuste de pesos, esses neurônios estarão mais parecidos com a entrada oferecida;
- **linha 13:** terminada a classificação de todos os estados de tabuleiros B_k presentes na base, a taxa de aprendizagem deve ser ajustada baseada no decremento estabelecido no começo do algoritmo (5%), ou seja, alpha = alpha * 0.05;
- **linha 14:** ajustada a taxa de aprendizagem, testa-se a condição de parada do algoritmo, de modo que, se ela não tiver sido atingida, os passos de 4 a 12 serão executados novamente. Caso a taxa de aprendizagem já tenha sido atingida, a rede estará apta a classificar corretamente os estados da base. A partir desse momento, não serão mais efetuadas quaisquer alterações nos pesos da rede e ela passa da fase de treinamento para a fase de utilização efetiva. O processo de classificação e geração dos arquivos com os tabuleiros classificados acontece da linha 16 a linha 22 do algoritmo;
- **Linha 16:** para cada estado de tabuleiro B_k presente na base de dados, as linhas de 17 a 19 são executadas na intenção de se descobrir qual neurônio de saída melhor representa o estado.
- **linha de 17 a 19:** dado um estado B_k, é calculada a distância entre tal estado e cada neurônio de saída da rede no intuito de descobrir a qual neurônio de saída o estado B_k deve pertencer. A fórmula usada para verificar a qual neurônio o estado deve ser relacionado é similar a apresentada na linha 6 desse algoritmo. A diferença é que, depois de encontrado o neurônio que representará o estado, tanto o peso de tal neurônio como os pesos de seus vizinhos não são reajustados. Os pesos dos neurônios de saída devem ser reajustados somente durante o processo de treinamento da rede;
- **linha 20:** dado que já tenham sido calculadas as distâncias entre o tabuleiro de entrada B_k e todos os neurônios de saída, o neurônio de saída Y_j "vencedor"que

representará o estado de tabuleiro será o que possuir o menor valor do quadrado da distância Euclidiana D_j;

- **linha 21:** nesta linha do algoritmo, o estado de tabuleiro B_k é armazenado no *cluster* Y_j. Quando o estado B_k é direcionado para o *cluster*, sua representação por característica é abandonada, sendo que o estado armazenado no *cluster* é o estado de representação vetorial. Não se esqueça que o estado de tabuleiro representado por característica é usado somente para definir em qual *cluster* um dado estado deve ser armazenado, como explicado na seção 4.1.1.

Concluída a distribuição dos 4000 estados de tabuleiro de final de jogo entre os 25 *clusters* a fim de limitar a quantidade de de estados de cada *cluster* (de modo a reduzir o tempo de treinamento dos agentes de final de jogo) efetua-se um processo que seleciona os 25 estados primeiros estados presentes em cada *cluster*, sendo que ao final desse processo cada *cluster* deve conter apenas 25 estados.

Tão logo o processo de *clusterização* é concluído, associa-se uma rede MLP com arquitetura similar a do *IIGA* a cada um dos 25 *clusters* gerados pela rede *Kohonen-SOM*. Tais redes são treinadas de modo a se especializar para tabuleiros de final de jogo que representam (definidos pelos *clusters*).

5.4.3 Treinamento das Redes Multiagentes de Final de Jogo

A fase de treinamento dos agentes de final de jogo que compõem o *MP-Draughts* é análoga à fase de treinamento do *VisionDraughts* e do *IIGA*. A diferença é que as redes de final de jogo são treinadas a partir dos 25 estados de tabuleiros de final de jogo presentes no *cluster* que elas representam.

Os pesos da rede neural que representam um agente de final de jogo são reajustados durante a fase do treinamento por *self-play* com clonagem por meio da técnica de aprendizagem por Diferenças Temporais TD(λ). O processo de treinamento, detalhado a seguir, acontece para cada um dos agentes acoplados aos *clusters* de estados de tabuleiro de final de jogo, descritos na seção anterior.

A ideia do treinamento por *self-play* com clonagem é treinar o jogador *opp1* por vários jogos contra uma cópia de si próprio, o *opp1-clone*. Os jogos de treinamento devem percorrer todos os 25 estados de tabuleiros, sendo realizados 200 jogos em cada estado de tabuleiro, totalizando 5000 jogos de treinamento. Desses 200 jogos,o agente joga 100 deles com peças pretas e 100 com peças vermelhas. Observe que essa estratégia de treinamento utilizada pelos agentes que compõem *MP-Draughts* é eficiente, uma vez que o agente jogador *opp1* deve sempre procurar melhorar o seu nível de desempenho a cada sessão de treinamento de forma a poder bater seu clone *opp1-clone*. Durante a fase de treinamento somente a rede *opp1* tem seus pesos reajustados.

No final de uma bateria de 5000 jogos de treinamento, acontece um torneio entre o agente com pesos reajustados *opp1* e seu clone *opp1-clone*. São realizados dois jogos em cada estado de tabuleiro, num total de 50 jogos-teste realizados. Dos dois jogos-teste que acontecem em cada estado de tabuleiro, no primeiro deles, o agente *opp1* joga com as peças pretas do tabuleiro de Damas e o agente *opp1-clone* joga com as peças vermelhas. Já no segundo jogo, as posições de ambos os agentes são invertidas. O objetivo dessa troca de posições dos jogadores sobre o tabuleiro é permitir uma melhor avaliação do desempenho dos agentes *opp1* e *opp1-clone* ao jogarem entre si em ambos os lados do tabuleiro, uma vez que as características se referem a restrições sobre peças pretas e/ou vermelhas.

Ao final dos 50 jogos-teste, se o agente *opp1* conseguir ganhar no mínimo 60% dos jogos, faz-se uma nova clonagem e o jogador passa a treinar contra esse novo clone. O processo se repete por uma bateria de 10 sessões de jogos de treinamento e, no final dessas baterias, realiza-se um torneio entre todos os clones obtidos no decorrer do treinamento. O jogador vencedor desse torneio é eleito para representar o *cluster* ao qual ele é acoplado.

5.4.4 Escolha do *EGA*

Após o treinamento dos agentes de final de jogo (processe descrito na seção 5.4.3), o *MP-Draughts* contará com 26 agentes treinados para jogar Damas, o *IIGA* e os 25 agentes especializados em fase de final de jogo.

A dinâmica de um jogo de Damas envolvendo o *MP-Draughts* acontece da seguinte maneira: o *IIGA* é o jogador que inicia o jogo de Damas e o conduz até que o tabuleiro tenha, no máximo, 13 peças. A partir desse momento, um dos agentes especialistas em final de jogo é que deve assumir o jogo, ou seja, esse agente deve substituir o *IIGA* na partida e conduzi-la até que ela termine. O agente de final de jogo escolhido entre os 25 agentes especialistas para finalizar o jogo - aqui chamado *EGA* - é aquele que melhor representa o estado de final de jogo corrente.

Faz-se o processo de escolha do *EGA* do seguinte modo: o estado de tabuleiro de final de jogo corretente é submetido à rede *Kohonen-SOM* treinada. Detecta-se o neurônio de saída Y_j que apresenta a menor distância Euclidiana com relação àquele estado. O agente de final de jogo que estiver associado ao *cluster* definido por Y_j será escolhido como *EGA*, conforme pseudo-código apresentado e comentado a seguir.

Pseudo-código referente a fase de seleção do agente de final de jogo - *EGA* - responsável por finalizar a partida

```
1: Select-EGA(board:board-state)
2: for each EndGameNetwork in set of EndGameNetwork: do
3:    d[i] = distancia(board-state, EndGameNetwork[i])
```

4: **end for**
5: j = i of minimum(d[i])
6: return j

Onde:

- **linha 1**: o algoritmo *Select-EGA* recebe, como argumento, o estado corrente do tabuleiro de jogo de Damas, *board-state*;
- **linha 2**: para cada um dos agentes especializados em final de jogo que compõe o *MP-Draughts*, executa-se a linha 3 do algoritmo, de modo a encontrar o agente que substituirá o *IIGA* no jogo;
- **linha 3**: calcula-se o quadrado da distância Euclidiana entre o estado corrente do jogo (tabuleiro de final de jogo) e cada um dos 25 agentes de final de jogo. O valor resultante desse cálculo é armazenado na variável *d[i]*. Para cada um dos agentes de final de jogo, calcula-se o quadrado da distância Euclidiana entre as características presentes no tabuleiro corrente e as características presentes no agente em questão;
- **linha 5**: depois de calculado o quadrado da distância Euclidiana entre o estado corrente e os agentes de final de jogo, verifica-se qual das distâncias é a menor. O valor da variável i correspondente à posição onde se encontra a menor distância, e indica o agente de final de jogo *EGA* que deverá ser selecionado;
- **linha 6**: depois de definido o *EGA*, o valor referente a ele é retornado para a função principal. A função principal transfere o jogo para ele.

Até o momento, apresentaram-se todos os agentes que compõem o *MP-Draughts* (*IIGA* e *EGA*), assim como os passos necessários para se obter cada um desses agentes. Na próxima seção, mostra-se é apresentado como é feita a interação desses agentes durante um jogo de Damas.

5.5 Resultados Experimentais

Esta seção apresenta a eficiência da arquitetura multiagente do *MP-Draughts* comparada com as arquiteturas do *VisionDraughts* [11] e do *NeuroDraughts* [42]. Os parâmetros levados em consideração na comparação foram o número de vitórias, empates e derrotas obtidos por cada agente em um torneio de 20 jogos. É interessante ressaltar que todos os jogadores tiveram o mesmo tipo de treinamento (*self-play* com clonagem) e o mesmo número de jogos de treinamento (5000 jogos de treinamento, seção 5.4.3). A igualdade no número de treinamento foi mantida no sentido de não favorecer ou prejudicar qualquer um dos jogadores.

Jogador	Tempo Treinamento (min)	Tempo Treinamento (%)
VisionDraughts	9,0	14%
MP-Draughts (média 25 agentes)	63,6	100%

Tabela 5.2: Tempo de treinamento dos jogadores *VisionDraughts* e *MP-Draughts*

As seções seguintes apresentam os resultados obtidos pelo *MP-Draughts* no que se refere a: tempo de treinamento, impacto de arquiteturas multiagentes e, além disso, mostra os bons resultados conseguidos pelo *MP-Draughts* nas situações de *loop* de final de jogo. Para avaliar o desempenho do *MP-Draughts* no que diz respeito a frequência de *loop* de final de jogo, realizou-se um torneio entre ele e os jogadores *VisionDraughts* e *NeuroDraughts*.

Conforme mostrado, as ocorrências de *loop* de final de jogo envolvendo o *MP-Draughts* restringem-se a 0% jogando contra o *NeuroDraughts*, e a 5% jogando contra o *VisionDraughts*. A título de comparação, a frequência de ocorrência de *loop* em jogos entre o *NeuroDraughts* e o *VisionDraughts* é de 25%.

5.5.1 Impacto Causado no Tempo de Treinamento

O módulo de busca eficiente em árvores de jogos desenvolvido no *VisionDraughts* (apresentado no capítulo 4) forneceu ao jogador automático maior agilidade no processo de treinamento. Com essa agilidade, foi possível treiná-lo em 5000 jogos durante aproximadamente, 9 minutos, como mostra a tabela 5.2.

O treinamento do agente *MP-Draughts*, mais especificamente, o treinamento dos 26 agentes que o compõem, utilizou o mesmo módulo de busca eficiente do *VisionDraughts*. O tempo dedicado para o treinamento do *IIGA* foi o mesmo dedicado ao treinamento do *VisionDraughts*, aproximadamente, 9 minutos. Já o tempo dedicado para o treinamento dos 25 agentes especialistas em final do jogo foi de aproximadamente, 63 minutos, segundo a tabela 5.2.

O aumento do tempo de treinamento deu-se, principalmente, porque movimentos executados nas fases finais de jogos de Damas precisam ser muito bem avaliados, uma vez que um erro nessas fases tendem a levar a derrotas. Em razão dessa característica de final de jogo, o tempo gasto para executar um movimento em um tabuleiro de final de jogo também é maior do que o tempo gasto para executar um movimento nm tabuleiro de início de jogo.

Note, na tabela 5.2, que o *VisionDraughts* é treinado em apenas 14% do tempo de treinamento do *MP-Draughts*. Logo, pode-se concluir que o aprendizado de jogadores especialistas em fases específicas de jogo demora mais tempo para convergir do que o aprendizado daqueles concebidos para atuar em todas as fases do jogo. Apesar disso, a

	MP-Draughts x VisionDraughts	MP-Draughts x NeuroDraughts
Vitórias	13	14
Derrotas	3	3
Empates	4	3
Loops	1	0

Tabela 5.3: Resultado de torneio de 20 jogos entre: *MP-Draughts*, *VisionDraughts* e *NeuroDraughts*

estratégia multiagentes apresenta uma boa alternativa, uma vez que ela se mostrou mais eficiente, como pode ser comprovado nos resultados apresentados nas próximas seções.

5.5.2 Impacto da Criação de Agentes Especializados em Fases Distintas do Jogo de Damas

As habilidades que um jogador tem de ter para finalizar um jogo com sucesso são diferentes das habilidades necessárias para executar boas jogadas no início de um jogo. Nessa linha, o *MP-Draughts* conta com um agente de início de jogo com habilidades suficientes para executar boas jogadas no início do jogo e com agentes de final de jogo com habilidades essenciais para final de jogos.

Ao passo que no *MP-Draughts* a capacidade de desenvolver habilidades específicas em cada fase do jogo de Damas é provida por agentes especializados em fases distintas do jogo de Damas, o que possibilita ao campeão mundial de Damas *Chinook* [71] manter sua habilidade ao longo das fases do jogo são 5 funções de avaliação ajustadas manual e exaustivamente por especialistas humanos. Observe que o jogador *MP-Draughts* conseguiu desenvolver habilidades em fases distintas de um jogo, assim como *Chinook*, porém, com o mínimo de intervenção humana possível, contando, na maior parte do tempo, com técnicas inteligentes para desenvolver essas habilidades.

Para verificar o impacto da criação dos agentes especializados, realizo-se um torneio de 20 jogos entre o jogador *MP-Draughts* e os jogadores *VisionDraughts* e *NeuroDraughts*. Os resultados do torneio, mostrados na tabela 5.3, comprovam a eficiência de sistemas jogadores multiagentes comparados com sistemas baseados em agentes simples.

Como é possível verificar na tabela 5.3, no torneio entre o *MP-Draughts* e o *Vision-Draughts*, o primeiro ganhou 65% dos jogos, empatou 20% e perdeu apenas 15%. Desses 20% de empate, somente 5% foram causados por problemas de *loop*. Já no torneio entre *MP-Draughts* e o *NeuroDraughts*, o *MP-Draughts* ganhou 70% , empatou 15% e perdeu 15%, não havendo nenhum empate causado por *loop* de final de jogo.

Como trabalho futuro, os autores deste trabalho pretendem desenvolver uma interface apropriada que permita ao *MP-Draughts* jogar com o campeão mundial, *Chinook*, no sentido de avaliar o desempenho do *MP-Draughts*.

	MP-Draughts x VisionDraughts	MP-Draughts x NeuroDraughts	VisionDraughts x NeuroDraughts
Empates	4	3	9
Empates - *loop*	1	0	5

Tabela 5.4: Número de empates causados por *loop* em 20 jogos entre os jogadores *MP-Draughts*, *VisionDraughts* e *NeuroDraughts*

5.5.3 Impacto da redução dos casos de *loop*

Com a diminuição dos casos de *loops* nos torneios envolvendo jogadores automáticos, a eficiência desses jogares é significativamente melhorada, o que pode ser constatado nos resultados apresentados na seção anterior.

Tanto o *NeuroDraughts* quanto o *VisionDraughts* mostraram-se ineficientes para finalizar diversas partidas do jogo de Damas, justamente, por causa da frequente ocorrência de *loop* de final de jogo, fato que pode ser observado na última coluna da tabela 5.4 que apresenta resultados de empates em torneio de 20 jogos envolvendo ambos os jogadores. No caso, o *VisionDraughts* terminou **9** partidas em empate contra o *NeuroDraughts*, sendo **5** delas causadas por *loop* de final de jogo. As ocorrências de *loop* de final de jogo, nesse caso, corresponderam a 25% do total de jogos.

Já nos jogos envolvendo o *MP-Draughts*, os casos de *loop* de final de jogo foram consideravelmente reduzidos. A redução dos casos de *loops* deu-se porque, como abordado anteriormente, o *MP-Draughts* conta com agentes especialistas em fases distintas do jogo de Damas. Esses agentes possuem habilidades mais refinadas para decidir qual jogada executar em um determinado momento do jogo, tornando-os mais ofensivos e mais conhecedores dessa fase do jogo.

Pode-se concluir que jogadores especializados em final de jogo têm mais conhecimento do ambiente e, consequentemente, mais conhecimento para executar uma jogada que tire-o de uma situação de *loop* e conduza-o para a "vitória óbvia". A tabela 5.4 apresenta também os casos de empate que ocorreram nos 20 jogos entre *MP-Draughts* x *VisionDraughts* e *MP-Draughts* x *NeuroDraughts*.

Analisando os casos de empates acontecidos durante os torneios, foi possível verificar que, dos 4 empates ocorridos durante o torneio contra o *VisionDraughts*, apenas um foi por ocorrência de *loop*; os outros 3 foram empates reais do jogo. Já no torneio contra o *NeuroDraughts*, não se constatou nenhum caso de *loop*, o que comprova, novamente, que o *MP-Draughts* supera seus predecessores em fase de final de jogo.

A título de esclarecimento, em Damas, empates reais são situações em que os jogadores encontram-se em equilíbrio no jogo, e a configuração do tabuleiro não deixa alternativa para que se dê um desfecho ao jogo após uma quantidade limite de jogadas. Exemplos de configuração de tabuleiros que são considerados empates reais:

- tabuleiros com 2 rainhas contra 2 rainhas;
- tabuleiros com 2 rainhas contra 1 rainha;
- tabuleiros com 2 rainhas contra 1 rainha e uma peça simples.

Concluindo, o uso de agentes especializados em fases distintas do jogo contribuiu para diminuir, consideravelmente, o número de partidas encerradas indevidamente em *loop* durante o torneio, o que impactou, diretamente, na eficiência do *MP-Draughts*.

Capítulo 6

Conclusões e Propostas Futuras

Apresentou-se neste trabalho, o *MP-Draughts*, um agente multiagente jogador de Damas que conta com agentes especializados em fases distintas do jogo. Esse jogador possui 26 agentes especializados, sendo: um agente especializado nas fases iniciais/ intermediárias de um jogo (tabuleiros com no mínimo 13 peças); e outros 25 agentes especializados em fase de final de jogo (tabuleiros com no máximo 12 peças). Os agentes jogadores que compõem o *MP-Draughts* correspondem a redes neurais MLPs treinadas pelo método de aprendizagem por reforço TD(λ), aliado a um eficiente método de busca baseado em poda alfa-beta com tabela de transposição e aprofundamento iterativo. O treinamento desses jogadores acontece por *self-play* com clonagem e as entradas das redes neurais correspondem aos estados de tabuleiros de Damas representados por um conjunto de características. O *MP-Draughts* conta com um mínimo possível de intervenção humana no seu processo de aprendizagem, ao contrário do atual campeão do mundo, *Chinook*, que conta com ela no ajuste manual das suas funções de avaliação e no uso de duas bases de dados existentes nele(de início e fim de jogo).

O problema do *loop* de final de jogo identificado nos jogadores *NeuroDraughts*, *LS-Draughts* e *VisionDraughts* foi a principal motivação que levou à criação do jogador multiagente e, além disso, é claro, o fato de se desejar obter jogadores especializados mais ofensivos em fases distintas do jogo de Damas.

Os resultados experimentais obtidos de torneios entre o *MP-Draughts* e os jogadores *NeuroDraughts* e *VisionDraughts* mostram que a ocorrência do problema do *loop* foi, consideravelmente, reduzida. A utilização de agentes especializados em fases distintas de um mesmo jogo possibilitou ao jogador o aumento na sua capacidade para tomar melhores decisões, uma vez que o ele possui uma visão mais refinada de cada fase do jogo. Quanto menor o ambiente em que o jogador deve agir, mais detalhes ele consegue enxergar dentro desse ambiente.

De forma geral o jogador multiagente mostrou-se mais eficiente na redução do problema de *loop* e desempenho no jogo que os jogadores precedentes a ele. Isso que pode ser observado nos resultados apresentados na seção 5.5.

Assim sendo, a proposta de trabalho apresentada e os resultados alcançados se mostraram bem promissores. A grande vantagem desse tipo de jogador é sua autonomia no processo de aprendizado, restringindo o uso de recursos externos apenas ao conjunto de características de Samuel [61], [62]. Já a principal desvantagem deste jogador foi o tempo dedicado ao treinamento do mesmo. Como pode ser observado na tabela 5.2 na seção 5.5, o tempo de treinamento do *MP-Draughts* é aproximadamente 450% maior do que o tempo de treinamento de um agente simples.

6.1 Perspectiva de Trabalhos Futuros

Apesar do bom desempenho do *MP-Draughts*, muitas técnicas ainda podem ser aplicadas em sua arquitetura na intenção de obter melhores resultados. Algumas sugestões de técnicas a serem aplicadas estão relacionadas abaixo:

- integrar ao *MP-Draughts* o módulo de algoritmo genético desenvolvido por Neto e Julia [50], [51], [52] no jogador *LS-Draughts* a fim de descobrir qual é o conjunto mínimo de características fundamentais para representar final de jogo. Atualmente, essa escolha é feita manualmente;
- desenvolver uma interface apropriada que permita ao *MP-Draughts* jogar com o campeão mundial, *Chinook*, no sentido de verificar o desempenho desse jogador quando comparado com o melhor do mundo;
- encontrar outras técnicas para representar os estados de tabuleiros de final de jogo no momento da *clusterização* desses estados por exemplo, foi feita uma primeira tentativa de uso de representação vetorial do estado de tabuleiro, porém esta se mostrou pior do que a *clusterização* usando características. Então, sugere-se a criação de algumas heurísticas no sentido de melhorar a *clusterização* feita em estados representados de forma vetorial;
- utilizar técnicas paralelas no processo de busca pelo melhor movimento (algoritmos paralelos), principalmente, considerando a popularização dos processadores com vários núcleos de processamento. Boas referências sobre paralelismo de algoritmos podem ser encontrada em [58], [66], [32];
- utilizar técnicas de processamento distribuído na etapa de treinamento.

Referências Bibliográficas

[1] A. Jain, R. D. *Algorithms for clustering data.* Prentice-Hall, 1988.

[2] A. M. Yip, C. Ding, T. F. C. Dynamic cluster formation using level set methods. *IEEE Transactions on PatternAnalysis and Machine Intelligence 28*, 6 (2006), 877–889.

[3] B. Moulin, B. C.-D. *An Overview of Distributed Artificial Intelligence - Foundations of distributed artificial intelligence.* John Wiley and Sons, 1996.

[4] B. S. Everitt, S. Landau, M. L. *Cluster analysis.* Arnold Publishers, 2001.

[5] Bartels, A., and Zeki, S. The theory of multistage integration in the visual brain. *Proceedings of the Royal Society* (1993).

[6] Baxter, J., Tridgell, A., and Weaver, L. Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal 21*, 2 (1998b), 84–99.

[7] Baxter, J., Trigdell, A., and Weaver, L. Knightcap: a chess program that learns by combining TD(λ) with game-tree search. In *Proc. 15th International Conf. on Machine Learning* (1998a), Morgan Kaufmann, San Francisco, CA, pp. 28–36.

[8] Bittencourt, G. *Inteligência Artificial Ferramentas e Teorias.* Editora da UFSC, 2006.

[9] Breuker, D., Uiterwijk, J., and van den Herik, H. Information in transposition tables, 1997.

[10] Breuker, D., Uiterwijk, J., and van den Herik, H. J. Replacement schemes for transposition tables, 1994.

[11] Caexêta, G. S. Visiondraughts - um sistema de aprendizagem de jogos de damas baseado em redes neurais, diferenças temporais, algoritmos eficientes de busca em Árvores e informações perfeitas contidas em bases de dados. Dissertação(mestrado em ciência da computação), Faculdade de Computação - Universidade Federal de Uberlândia, Uberlândia, Brasil, 2008.

[12] Caixeta, G. S., and Julia, R. M. S. A draughts learning system based on neural networks and temporal differences: The impact of an efficient tree-search algorithm. *The 19th Brazilian Symposium on Artificial Intelligence, SBIA* (2008).

[13] Campos, P., and Langlois, T. Abalearn: A risk-sensitive approach to self-play learning in abalone. In *Proceedings of the European Conference on Machine Learning* (2003), pp. 35–46.

[14] Chellapilla, K., and Fogel, D. B. Anaconda defeats hoyle 6-0: A case study competing an evolved checkers program against commercially available software. In *Proceedings of the 2000 Congress on Evolutionary Computation CEC00* (La Jolla Marriott Hotel La Jolla, California, USA, 6-9 2000), IEEE Press, pp. 857–863.

[15] Chellapilla, K., and Fogel, D. B. Evolving an expert checkers playing program without using human expertise. *IEEE Trans. Evolutionary Computation 5*, 4 (2001), 422–428.

[16] Darwen, P. J. Why co-evolution beats temporal difference learning at backgammon for a linear architecture, but not a non-linear architecture. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001* (COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 2001), IEEE Press, pp. 1003–010.

[17] F. Zambonelli, R. N. Jennings, M. W. Organisational abstractions for the analysis and design of multi-agent systems. In *Procedings of the 1st International Workshop on Agent-Oriented Software Engineering* (2000).

[18] Fausett, L. *Fundamentals of Neural Networks: Architectures, Algorithms And Applications*. Prentice Hall, 1994.

[19] Fogel, D. B., and Chellapilla, K. Verifying anaconda's expert rating by competing against chinook: experiments in co-evolving a neural checkers player. *Neurocomputing 42*, 1-4 (2002), 69–86.

[20] Fogel, D. B., Hays, T. J., Hahn, S. L., and Quon, J. A self-learning evolutionary chess program. *Proceedings of the IEEE 92*, 12 (2004), 1947–1954.

[21] Freeman, W. J. The physiology of perception. *Scientific American* (1991).

[22] Frey, P. W. *Chess Skill in Man and Machine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1979.

[23] Gilbert, E. Kingsrow, 2008. Disponível em: <http://pages.prodigy.net/eyg/Checkers/KingsRow.htm>.

[24] Hartmann, C. Linguagem e ferramenta de autoria para promover o desenvolvimento de perícis em xadrez. Master's thesis, Universidade Federal do Paraná, 2005.

[25] Haykin, S. *Redes Neurais: Princípios e Prática (2º edição)*. Bookman Editora, Porto Alegre, RS, 2001.

[26] Holland, J. H. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[27] Hubel, D., and Wiesel, T. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *Journal of Physiology* (1962).

[28] Jain, A., and Dubes, R. *Algorithms for clustering data*. Prentice-Hall, 1988.

[29] Jennings, N. R., Ed. *Agent-Oriented Software Engineering. In: Proceedings of the 9Th European Workshop on Modelling Autonomous Agent in a Multi-Agent World: Multi-Agent System Engineering* (1999), vol. 1647 of *Lecture Notes in Computer Science*, Springer-Verlag.

[30] Jennings, N. R., Faratin, P., Johson, J. M., O'Brien, P., and Wiegand, M. E. Using intelligent agents to manage business processes. *Proceedings of the International Conference on PRatical Applications of Intelligent Agents and Multi-AgentsTechnology - (PAAM-96)* (1996).

[31] Jennings, N. R., and Wooldridge, M. J. *Applications of intelligent agents - Agent technology: foundations, applications, and markets*. Springer-Verlag, 1998.

[32] Kishimoto, A., and Schaeffer, J. Distributed game-tree search using transposition table driven work scheduling. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 323.

[33] K.Lagus, Honkela, T., Kaski, S., and Kohonen, T. Websom for textual data mining. *Artificial Intelligence Review 13* (1999), 345–364.

[34] Kohonen, T. Analysis of a simple self-organizing process. *Biological Cybernetics* (1982), 135–140.

[35] Kohonen, T. Self-organizing formation of topologically correct feature maps. *Biological Cybernetics* (1982), 59–69.

[36] Kohonen, T. *Self-Organization and Associative Memory*, 3rd ed. 1989 ed. Springer, 1984.

[37] Kohonen, T. The self-organizing map. *Proc. IEEE* (1990), 1464–1480.

[38] Kohonen, T. *Self-Organizing Maps*. Springer, 2001.

[39] Kovacs, Z. L. *Redes Neurais Artificiais: Fundamentos e Aplicações*. São Paulo: Collegium Cognitio, 1996.

[40] Lake, R., Schaeffer, J., and Lu, P. Solving large retrograde analysis problems using a network of workstations, 1994. Disponível em: http://citeseer.ist.psu.edu/lake94solving.html.

[41] Leuski, A. Learning of position evaluation in the game of othello. Tech. rep., University of Massachusetts at Amherst, Amherst, MA, January 1995.

[42] Lynch, M. Neurodraughts: An application of temporal difference learning to draughts. Master's thesis, Department of Computer Science and Information Systems, University of Limerick, Ireland, 1997.

[43] Lynch, M., and Griffith, N. Neurodraughts: the role of representation, search, training regime and architecture in a td draughts player. *Eighth Ireland Conference on Artificial Intelligence* (1997), 64–72. Disponível em: <http://iamlynch.com/nd.html>.

[44] M. Wooldridge, N. R. J. Intelligent agents: Theory and practice. *The Knowledge Engineering Review 10*, 2 (1995).

[45] Marsland, T. A. A review of game-tree pruning. *ICCA Journal 9*, 1 (1986), 3–19.

[46] McCulloch, W., and Pitts, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics 5* (1943), 115–133.

[47] Millington, I. *Artificial Intelligence for Game.* Morgan Kaufmann, 2006.

[48] Mitchell, T. M., and Thrun, S. Explanation based learning: A comparison of symbolic and neural network approaches. *Tenth International Conference on Machine Learning* (1993).

[49] Mitchell, T. M., and Thrun, S. Explanation-based neural network learning for robot control. *Advances in Neural Information Processing Systems 5* (1993).

[50] Neto, H. C. Ls-draughts - um sistema de aprendizagem de jogos de damas baseado em algoritmos genéticos, redes neurais e diferenças temporais. Dissertação(mestrado em ciência da computação), Faculdade de Computação - Universidade Federal de Uberlândia, Uberlândia, Brasil, 2007.

[51] Neto, H. C., and Julia, R. M. S. Lcs-draughts-um sistema automático classificador e gerador de características em jogos de damas., 2007.

[52] Neto, H. C., and S.Julia, R. M. Ls-draughts - a draughts learning system based on genetic algorithms, neural network and temporal differences. In *IEEE Congress on Evolutionary Computation* (2007), IEEE, pp. 2523–2529.

[53] Oliveira, T. B. S. Clusterização de dados utilizando técnicas de redes complexas e computação bioinspirada. Master's thesis, Instituto de Ciências Matemáticas e de Computação - ICMC/USP, 2008.

[54] Patist, J.-P., and Wiering, M. Learning to play draughts using temporal difference learning with neural networks and databases.

[55] Plaat, A. *Research Re: search & Re-search.* PhD thesis, Rotterdam, Netherlands, 1996.

[56] Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. Best-first fixed-depth game-tree search in practice. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)* (Montreal, Canada, 1995), vol. 1, pp. 273–279.

[57] Pollack, J. B., and Blair, A. D. Co-evolution in the successful learning of backgammon strategy. *Machine Learning 32*, 1 (1998), 225–240.

[58] Rasmussen, D. Parallel chess searching and bitboards. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2004. Supervised by Prof. Jens Clausen.

[59] Ribeiro, C. H. C., and Monteiro, S. T. Aprendizagem da navegação em robôs móveis a partir de mapas obtidos autonomamente. In *IV Encontro Nacional de Inteligência Artificial (ENIA)* (2003).

[60] Russell, S., and Norvig, P. *Inteligência Artificial - Uma Abordagem Moderna (2a edição).* Editora Campus, 2004.

[61] Samuel, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development 3*, 3 (1959), 211–229.

[62] Samuel, A. L. Some studies in machine learning using the game of checkers ii - recent progress. *IBM Journal of Research and Development 11*, 6 (1967), 601–617.

[63] Schaeffer, J. Man versus machine: The silicon graphics world checkers championship, 1992.

[64] Schaeffer, J. Applying the experience of building a high performance search engine for one domain to another, 2002.

[65] Schaeffer, J., Björnsson, Y., Burch, N., Lake, R., Lu, P., and Sutphen, S. Building the checkers 10-piece endgame databases. H. J. van den Herik, H. Iida, and E. A. Heinz, Eds., vol. 263 of *IFIP*, Kluwer, pp. 193–210.

[66] Schaeffer, J., and Brockington, M. G. APHID: Asynchronous parallel game-tree search. *Journal of Parallel and Distributed Computing 60*, 2 (2000), 247–273.

[67] Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Lu, P., and Sutphen, S. Checkers is solved. *Science* (2007), 1144079+.

[68] Schaeffer, J., Hlynka, M., and Jussila, V. Temporal difference learning applied to a high performance game-playing program. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (2001), 529–534.

[69] Schaeffer, J., L., R., Lu, P., and Bryant, M. Chinook: The world man-machine checkers champion. *AI Magazine 17*, 1 (1996), 21–29.

[70] Schaeffer, J., and Lake, R. Solving the game of checkers, 1996. Disponível em: <http://citeseer.ist.psu.edu/6903.html.

[71] Schaeffer, J., Lake, R., Lu, P., and Bryant, M. CHINOOK: The world man-machine checkers champion. *AI Magazine 17*, 1 (1996), 21–29.

[72] Schaeffer, J., and Plaat, A. New advances in alpha-beta searching. *Conference on Computer Science* (1996), 124–130.

[73] Schraudolph, N. N., Dayan, P., and Sejnowski, T. J. Learning to evaluate go positions via temporal difference methods. In *Computational Intelligence in Games Studies in Fuzziness and Soft Computing* (2001), I. Baba and Jain, Eds., vol. 62, Springer Verlag.

[74] Shams, R., Kaindl, H., and Horacek, H. Using aspiration windows for minimax algorithms. In *IJCAI* (1991), pp. 192–197.

[75] Sutton, R. S. *Temporal credit assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts, Amherst, 1984.

[76] Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine Learning 3*, 1 (1988), 9–44.

[77] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[78] Tesauro, G. Practical issues in temporal difference learning. In *Advances in Neural Information Processing Systems* (1992), J. E. Moody, S. J. Hanson, and R. P. Lippmann, Eds., vol. 4, Morgan Kaufmann Publishers, Inc., pp. 259–266.

[79] Tesauro, G. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation 6*, 2 (1994), 215–219.

[80] Tesauro, G. Temporal difference learning and td-gammon. *Communications of the ACM 38*, 3 (1995), 19–23.

[81] Thrun, S. Learning to play the game of chess. *Advances in Neural Information Processing Systems 7* (1995).

[82] van den Herik, H. J., Uiterwijk, J. W. H. M., and van Rijswijck, J. Games solved: now and in the future. *Artificial Intelligence 134*, 1-2 (2002), 277–311.

[83] Walker, M. A. An application of reinforcement learning to dialogue strategy selection in a spoken dialogue system for email. In *Journal of Artificial Intelligence Research 12* (2000), pp. 387–416.

[84] Wiering, M. Multi-agent reinforcement learning for traffic light control. In *Proc. 17th International Conf. on Machine Learning* (2000), Morgan Kaufmann, San Francisco, CA, pp. 1151–1158.

[85] Willshaw, D. J., and von der Malsburg, C. How patterned neural connexions can be set up by self-organisation. *Proc Roy Soc B* (1976).

[86] Xing, L., and Pham, D. *Neural Networks for Identification, Prediction, and Control*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.

[87] Yovits, M. C., and Cameron, S. *Self-Organizing Systems*. Bookmark, 1960.

[88] Zobrist, A. L. A hashing method with applications for game playing. Tech. rep., University of Wisconsin, Wisconsin, 1969.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo